View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by RIT Scholar Works

Rochester Institute of Technology
RIT Scholar Works

Theses | Thesis/Dissertation Collections

1990

Parallelism in declarative languages

Catherine Eleftherios Chronaki

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Chronaki, Catherine Eleftherios, "Parallelism in declarative languages" (1990). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Rochester Institute of Technology
School of Computer Science and Technology

Parallelism in Declarative Languages

by

Catherine Eleftherios Chronaki

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Professor Stanislaw P. Radziszowski

Professor Andrew T. Kitchen

Professor Peter G. Anderson

December 1990

Parallelism in Declarative Languages

Catherine E. Chronaki

December 1990

SAMPLE statements to **authorize** or **deny** permission to reproduce an RIT thesis.

1. Title of thesis Parallelism in Declarative Languages

I Catherine E. Chronaki hereby **grant permission** to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date 1/7/91

2. Title of thesis _____

_____

I _____ **prefer to be contacted** each time a request for reproduction is made. I can be reached at the following address.

_____

_____

Date _____

3. Title of Thesis _____

_____

I _____ hereby **deny** permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part.

Date _____

rev. 1/89

Abstract

Imperative programming languages were initially built for uniprocessor systems that evolved out of the Von Neumann machine model. This model of storage oriented computation blocks parallelism and increases the cost of parallel program development and porting. Declarative languages based on mathematical models of computation, seem more suitable for the development of parallel programs.

In the first part of this thesis we examine different language families under the declarative paradigm: functional, logic, and constraint languages.

Functional languages are based on the abstract model of functions and λ-calculus. They were initially developed for symbolic computation, but today they are commonly used in numerical analysis and many other application areas. Pure lisp is a widely known member of this class.

Logic languages are based on first order predicate calculus. Although they were initially developed for theorem proving, fifth generation operating systems are written in them. Most logic languages are descendants or distant relatives of Prolog.

Constraint languages are related to logic languages. In a constraint language you define a program object by placing constraints on its structure and its behavior. They were initially used in graphics applications, but today researchers work on using them in parallel computation. Here we will compare and contrast the language classes above, locate advantages and deficiencies, and explain different choices made by language implementors.

In the second part of thesis we describe a front end for the CONSUL, a prototype constraint language for programming multiprocessors. The most important features of the front end are compact representation of constraints, type definitions, functional use of relations, and the ability to split programs into multiple files.

Μην ταξιδευεις με το νου χωρις να βλεπεις δρομο,
και μην νομιζεις πως χαρα θα νοιωσεις διχως πονο...
Κρητικη Μαντιναδα

Σε δυο μαυρα ματακια.

Acknowledgments

Dr. Douglas Baldwin introduced me to the wonderful world of parallel declarative languages and CONSUL and inspired my interest in the field.

My advisor Dr. Stanislaw Radziszowski, with his enlightening comments and careful guidance helped me develop this thesis.

Dr. Peter Anderson read early versions of this work and provided me with helpful comments and directions.

I would like to thank Dr. Radziszowski, Dr. Peter Anderson, and Dr. Andrew Kitchen for serving members of my committee.

I am grateful Rochester Institute of Technology for the graduate scholarships that financed my studies. I would also like to thank University of Rochester, for providing the resources to write this thesis.

My parents, Eleftherios and Anna Chronaki guided and supported my education from prime school to university. For that and much more they have done for me, I am grateful and proud.

Last but not least, I would like to thank Evangelos Markatos for endless discussions of the subject, repeated corrections of early manuscripts and unsurpassed emotional support.

C. Chronaki
December 8, 1990

Contents

Part I

Declarative Languages and Parallelism

Chapter 1

Introduction

1.1 Imperative Languages

Early languages reflected the structure of the underlying hardware. Most early computer architectures followed the Von Neumann model: Their basic components were the CPU, the main memory, and the bus connection between them. Programming was based on the fetch-execute cycle:

I. Fetch the instruction arguments from memory,
II. Execute the instruction,
III. If necessary store back in memory the result.

Data movement on the bus connection is sequential, and programming was organized around it. Programming was imperative: The programmer instructed the computer to execute a sequence of commands that solved the problem. Since then, a number of abstractions have been built on top of the basic fetch execute circle, but the programmer is always very much aware of the existence of storage (i.e. storage oriented programming).

Imperative programming on parallel machines is still storage oriented. The programming task though, is much more complicated. Multiple processing elements, PEs, interact with the global state of the machine storage concurrently and the programmer's task is to impose explicit control on these interactions (synchronize, coordinate and generally preserve the consistency). The explicit control of these interactions results in code that is difficult to debug and port.

The programmer has to monitor the behavior of all PEs to collect and process data from all entities that potentially modify the state of the machine. This is one of the reasons that current parallel programs are either not portable or present high variance in execution time between different machines.

The most common way to exploit multiprocessors today is to give the potential of parallelism to the programmer, together with a set of primitive parallel program constructs and data structures in the best case. The programmer who works on a parallel architecture has to consider hardware issues such as memory latency and interprocess communication cost vs. computation time of the subtasks that can be executed in parallel. These considerations will affect deeply the control structure of his program. Naturally enough they will also raise significantly the development and maintainance cost. If later the configuration of the machine changes or the program needs to be ported to another architecture, the whole trade-off needs to be reevaluated.

Language designers address this problem of "control" by designing imperative languages with loose forms of control structure that can expose all the potential parallelism. It is the job of the compiler now, to map all these fine forms of control efficiently on the target architecture[Gupta, 1986].

Backus [Backus, 1978] pointed out in his 1977 Turing Award Lecture, that Von Neumann languages are inherently inefficient because they are built around a single program statement: the *assignment statement.* The importance of this statement was great because it was made, by the designer of Fortran, and a person that influenced significantly the design of Algol. The assignment statement stores the value of an expression in a memory cell. This operation has the side-effect of altering the binding(value) for the variable on the left hand side (LHS) of the statement. It further introduces an execution *dependency* on the program: all updates of a given memory cell (variable) have to be performed in the sequence implied by the structure of the program.

Even with the philosophical problems of the assignment statement, there have been attempts to exploit parallelism implicitly in imperative programming languages like Fortran. Compilers which use elaborate techniques based on data flow and interporcedural analysis of the program produce efficient parallel executable code. The compiler group in Rice have reported significant results in building such compilers[Allen and Kennedy, 1987; Flatt and Kennedy, 1989; Carle *et al.*, 1987].

In a parallel environment the characteristics of the underlying machine are not easy to bare in mind. There are a lot of architectures with different characteristics. This implies that the standards for a programming language that can efficiently exploit different parallel architectures change: Higher emphasis is given in portability, conciseness, abstraction, ability for formal reasoning and expressiveness of the underlying model, than in exposition of the underlying implementation of control. Languages within the declarative paradigm are possible candidate for this role.

1.2 Declarative Languages

Declarative languages take a different approach to programming by separating the declarative "what" from the imperative "how". This separation places a new layer of abstraction on top of the traditional languages given that a declarative language is translated in an imperative one (i.e. assembly or C).

In this thesis we support that this level of abstraction is necessary to adequately exploit parallelism:

- *The underlying architecture of the multiprocessor is transparent to the declarative language programmer.*

  Nonprocedural (i.e. declarative) languages encourage the programmer to spend more time thinking about the problem and programming the algorithm, than programming the architecture. The programmer writes the program based on "what" the solution to the problem is, as opposed to "how" the control and assignment statements of the language are used to implement the algorithm. If the program is not time critical it is possible that the program is installed without further embellishments. The rest is done by the compiler.

- *The ability to exploit parallel algorithms is not lost.*

  Declarative languages may also give the programmer the ability to specify "how" the program should be executed, separately from the core program itself. This approach is explored by *Parafunctional programming* and general purpose constraint programming. The programmer adds annotations to a regular functional (constraint) program, which the compiler processes and embeds in the PE structure that will finally execute the program. The programmer might in turn change his annotations and try another configuration. This refinement process is most efficient and speeds up program development.

- *Declarative languages are based on a well defined mathematical notion.*

  It is more straightforward to reason about the correctness properties of declarative programs than imperative ones.

- *Abstraction from storage oriented programming.*

  The language being high level directs the programmer to think more of the dependencies inherent to the problem than the execution order.

Unfortunately the picture is not at all so bright. There are disadvantages and inefficiencies associated with declarative languages[Gelernter, 1986]:

- *Lack of effective translation and optimization tools.*

  In the past most declarative languages were interactive and thus interpreter based. The languages evolved without abandoning their dynamic characteristics placing obstacles to their effective compilation. Nowadays many researchers are working on compilers for declarative languages. The structure of a compiler for a declarative language is different from that of conventional ones, because both the structure of the program and its semantics are different. Issues like optimization and storage management are considerably different.

- *Lack of experience in general purpose programming.*

  The first declarative languages were interface languages or meta-languages, special purpose languages. Today the suitability of declarative languages for general purpose programming is tested. There are some striking examples like the fifth generation project in Japan, which aims to built a whole parallel system in Prolog. Still, many researchers have serious doubts on the efficiency of the resulting system. Large applications need to be built, which are readable and maintainable, before people accept the paradigm.

- *Declarative languages do not support as familiar a programming style, as imperative languages.*

  The world is dominated by imperative programming and if a programmer is used to program in one language it is difficult to change his practices and adopt a different paradigm. This is the reason why there are so many languages out there (more than 3000) and so few of them are widely known and used. Programmers have to be persuaded of the usefulness of the declarative paradigm before they adopt it.

In the following chapters, we will describe different declarative paradigms: functional languages, based on the notion of functions, logic languages, based on the notion of relation, and constraint languages based on the notion of constraints. We will go through their characteristics and their relative power to express common algorithms, in conjunction with representatives from each class. We will comment on trends, implementation techniques and problems associated with their efficient translation on parallel machines.

Chapter 2

Functional Languages

2.1 The Functional paradigm

In *Functional languages*[Field and Harrison, 1988; Bird and Wadler, 1988], the basic operation is function application. Functional languages are the closest to the imperative paradigm in the declarative family. Pure Lisp, the first functional programming language, is mostly known for its strong symbol manipulation abilities and its acceptance in the Artificial Intelligence community. If thinking of functional languages brings deeply parenthesized s-expressions in mind, this is not what functional languages are today.

A functional program is a function applied on the program input and the output of the program is the resulting value. No restriction applies on the form of the input or output values therefore, a functional program of medium size may perform very complex operations.

In imperative languages there is the notion of the global program state. The global program state is composed of the bindings of program variables and the state of the machine. This characteristic not only creates an execution bottleneck but also bases computation on side effects, namely the storage of values in memory locations.[Backus, 1974; Backus, 1978] Functional languages are side-effect free. No explicit notion of storage or global state is present. The program state is transformed by the execution of program statements: values are stored and retrieved from memory locations.

A characteristic property of functional languages is *referential transparency*: A language has referential transparency when the value of any expression in the language, depends only on the expression textual context, not on the computational history. In other words, a function applied on the same arguments will

always produce the same result.

Referential transparency is very useful in reasoning formally about programs. It is much easier to perform transformations of functional programs than imperative programs because function transformations are mathematically well defined. In an imperative program a name in a given context may have many different meanings(values) due to side effects(aliasing). Therefore the formal transformation of imperative programs is largely ad-hoc.

One consequence of referential transparency is that functional programs are *deterministic.* A program run on some input will always produce the same output. Determinism is desirable if the user wants to exploit parallelism in a program: No matter what the program evaluation order is, the program will always give the same result as long as it conforms to the dependencies introduced by function application.

On the other hand the language does not give the user the ability/flexibility to write nondeterministic programs. For example, there are cases when the determinacy of functional languages is really hard: Suppose we want to program an I/O controller, operating system facilities, or tasks with inherent the notion of nondeterminism. We have to pass the complete state of the task as an argument to all program functions so that we are able to use the information associated with it.

The approach of functional languages to the assignment statement is very interesting. According to what we said, the assignment statement with imperative semantics is forbidden in the functional paradigm because of the side-effects and the history sensitivity it introduces to the program. For example a subroutine in Fortran or any other static language may be made history sensitive, by using the value of an initialized local variable in consecutive calls to a subroutines. This means that you do not even need to have global variables to implement the notion of state.

Some functional languages like FP[Backus, 1978] have no assignment statement. Others like Id Nouveu[Arvind and Ekanadham, 1988] allow names as a shorthand for expressions. The latter approach allows a single binding of a given name to an expression within the same scope. Reading an uninitialized name results to blocking, until the name is initialized, while trying to update an assigned name results to a runtime error. These languages are known as *single assignment languages* and their efficient compilation and run time support is an active area of research[Schnorf and Ganapathi, 1989; Arvind and Ekanadham, 1988; Arvind *et al.*, 1989].

Imperative programming languages developed in bottom up fashion, in an attempt to abstract from the machine. In this process the ideas of storage and

computation state were never abandoned. In constant to this evolution process, functional and generally declarative languages were developed in a top down fashion and there are a lot of inefficiencies that we have to overcome for the elegance and simplicity of the functional model.

Pure Lisp[McCarthy, 1960] was the first programming language to follow into the functional paradigm. According to McCarthy, his work on Pure Lisp was motivated by the need to develop an algebraic list processing language for use in artificial intelligence. Pure Lisp uses λ-expressions to introduce unnamed functions and according to McCarthy it was not motivated by λ-Calculus. Nevertheless the main ideas of the functional paradigm were there. From that point on, however λ-Calculus and Lisp continued to evolve separately as Lisp started to adopt more and more imperative characteristics for the sake of efficiency.

λ-Caluclus developed in parallel with work on functional programming languages as the underlying mathematical formulation. Here we will describe the characteristics of λ-Calculus to expose the elegance and simplicity of the model underlying functional languages.

A interesting property of λ-Calculus that influenced the design of functional languages up to our days was the idea of *curried* functions: λ-Calculus is restricted to functions of one argument. Curry[Curry and Feys, 1958] adopted the elegant notation *(f x y)* to denote *((f x) y)*, a *curried* function of one argument, which would have been alternatively written as a function of two arguments *(f(x y))*. This is perfectly legitimate in a functional language and reflects the idea of functions as first class objects(i.e. a function may appear wherever a regular variable can).

2.1.1 λ-Calculus

λ-Calculus can be considered as the first functional language. In [Hudac, 1989], Paul Hudac surveys λ-Calculus and its variations as it developed in parallel with functional language theory.

λ-Calculus is a syntax for terms and a set of rewrite rules for transforming terms. Church defined it in an attempt to capture intuition about the behavior of functions. The terms of the calculus are called λ-expressions.

A λ-expression is defined as:

$$
\begin{array}{lcll}
(\lambda - expression)\ e & = & x & \mid identifier \\
 & & (e_1\ e_2) & \mid (function\ application) \\
 & & \lambda x.e & (\lambda\text{ -}abstraction)
\end{array}
$$

λ-abstractions capture the notion of a function, while function applications capture the notion of function application. λ-Calculus functions can be applied to themselves and this ability of self application (recursion) is what gives λ-Calculus its power. Turing proved[Turing, 1937] that λ-Calculus can compute the set of functions computable by the Turing machines.

The rewrite rules use the notion of substitution of a free variable in an expression to perform function application. The set of free variables $Free(e)$ in an expression e is defined as:

$$\begin{aligned} Free(x) &= \{x\} \\ Free(e_1\ e_2) &= Free(e_1) \cup Free(e_2) \\ Free(\lambda x.e) &= Free(e) - \{x\} \end{aligned}$$

Having defined the set of free variables in an expression we can define inductively the substitution $[e1/x]e2$ of all free occurrences of an identifier x in an expression e_2, by an expression e_1:

$$[e/x_i]x_j = \begin{cases} e, & \text{if } i = j \\ x_j, & \text{otherwise} \end{cases} \tag{2.1}$$

$$[e_1/x](e_2\ \ e_3) = ([e_1/x]e_2)\ \ ([e_1/x]e_3) \tag{2.2}$$

$$[e_1/x_i]\lambda x_j.e_2 = \begin{cases} \lambda x_j.e_2, & \text{if } i = j \\ \lambda x_j.[e_1/x_i]e_2, & \text{if } i \neq j \text{ and } x_j \notin Free(e_1) \\ \lambda x_\ell.[e_1/x_i]([x_\ell/x_j]e_2), & \text{if } x_\ell \notin Free(e_1) \cup Free(e_2), \\ & \quad \text{where} \\ & \quad \ell \neq j \text{ and } \ell \neq i \end{cases} \tag{2.3}$$

The last case refers to renaming: an identifier changes its name so as not to introduce a name conflict within the same scope. Now we are ready to introduce the three rewrite rules on λ-expressions:

- **α-conversion** (*renaming*):

$$\lambda x_i.e \Longleftrightarrow \lambda x_j.[x_j/x_i]e, where\ \ x_j \notin Free(e). \tag{2.4}$$

- **β-conversion** (*function application*):

$$(\lambda x.e_1)e_2 \Longrightarrow [e_2/x]e_1. \tag{2.5}$$

- η-**conversion**:

$$\lambda x.(e\ x) \Longrightarrow e,\quad if\quad x \notin Free(e). \tag{2.6}$$

The rewrite rules together with the equivalence rules, for reflexivity, symmetricity and equivalence induce a theory of reduction on λ-Calculus which has been shown to be consistent[Church, 1941].

Definition 1 *e_1 can be* **reduced** *to e_2 ($e_1 \Rightarrow e_2$), if e_2 can be derived from e_1 by zero or more β-conversion, η-conversion or α-conversions.*

When we have applied all the β-conversions and η-conversions that are possible, then we say that the λ-expression is in *normal form:*

Definition 2 *A λ-expression is in* **normal form** *if it cannot be further reduced using β-conversion, or η-conversion.*

The computation of an λ-expression in λ-Calculus corresponds to its reduction to normal form.

Example 1 Assuming the associativity $((e_1\ e_2\ e_3) = ((e_1\ e_2)\ e_3))$ of function application consider the following simple conversions:

1.

$$\lambda x.(\lambda y.x)z \overset{\beta-conversion(2.5)}{\Longrightarrow} [z/x](\lambda y.x) \overset{substitution(2.3,2.1)}{\Longrightarrow} \lambda y.z$$

2.

$$\begin{gathered}
\lambda x.((\lambda y.x)(\lambda x.x)x)y \overset{\beta-reduction(2.5)}{\Longrightarrow} \\
[y/x]((\lambda y.x)(\lambda x.x)x) \overset{substitution(2.2)}{\Longrightarrow} \\
([y/x]\lambda y.x)[y/x]((\lambda x.x)x) \overset{\alpha-conversion(2.4)}{\Longrightarrow} \\
(\lambda z.[y/x]([z/y]x))[y/x]((\lambda x.x)x) \overset{substitution(2.1,2.2)}{\Longrightarrow} \\
(\lambda z.[y/x]x)([y/x]\lambda x.x)([y/x]x) \overset{substitution2.1)}{\Longrightarrow} \\
(\lambda z.y)([y/x]\lambda x.x)([y/x]x) \overset{substitution(2.3)}{\Longrightarrow} \\
(\lambda z.y)(\lambda x.x)([y/x]x) \overset{substitution(2.3)}{\Longrightarrow} \\
(\lambda z.y)(\lambda x.x)y
\end{gathered}$$

Unfortunately not all λ expressions have a normal form, and furthermore as we shall see the conversion order affects whether the computation terminates or not.

Example 2 The following λ-expression has no normal form. The conversion process goes on for ever without any chance to reach a normal form:

$$(\lambda x.(x\ \ x))\ \ (\lambda x.(x\ \ x)) \overset{\beta-conversion(2.5)}{\Longrightarrow}$$

$$[\lambda x.(x\ \ x)/x](x\ \ x) \overset{substitution(2.3)}{\Longrightarrow}$$

$$\lambda x.(x\ \ x)\ \ \lambda x.(x\ \ x))\cdots$$

We can consider this as an infinite loop because no forward progress out of it, can be ever made!

Reduction as the reflexive transitive closure of the three rewrite rules(2.4, 2.5, 2.6) we introduced, corresponds to the *evaluation* of an expression.

There are expressions that do not have a normal form. Normal form gives a clear way to express finality(terminating computations) in the computational sense. Once a λ-expression has reached its normal form its computation is complete in the sense that no more reductions except for renaming may be performed. Note that this does not preclude the existence of expressions for which the reduction process does not terminate.

The following two theorems are very important for λ-Calculus and systems with analogous properties (called Church-Rosser systems):

Theorem 1 (Church-Rosser I) *[Church, 1941]:*
Let $\overset{*}{\Longleftrightarrow}$ *be the reflexive, transitive closure of* $\Longrightarrow$, *if* $e_0 \overset{*}{\Longrightarrow} e_1$, *then there exists an* e_2, *such that* $e_0 \overset{*}{\Longleftrightarrow} e_2$ *and* $e_1 \overset{*}{\Longleftrightarrow} e_2$.

Theorem 1 says that if two λ-expressions are mutually reducible then they can be both reduced to a third not necessarily distinct λ-expression.

The following corollary of theorem 1 is very important:

Corollary 1 *No λ-expression can be reduced to two distinct normal forms.*

So, irrespectively of the order we apply conversions on the different reducible parts of a λ-expression the normal form we will produce will be the same. This idea is important for parallel computation: we can potentially evaluate different parts of the program-expression in different processing elements and thus arrive faster at the result. This is why parallel execution models of functional languages are heavily based on efficient ways to execute parallel graph reduction. We can model the program as a graph and apply standard reduction techniques to assign different pieces of the graph to different processing elements.

However there is still a question to bother us:

Is it always possible to find the normal form of a λ-expression?

There are two basic methods of sequentially reducing a λ-expression:

1. **Normal Order Reduction**: The reduction proceeds left to right, and whenever there are more than one reducible expression we reduce left to right.

   Normal order reduction corresponds to the "call by name" argument passing mechanism in Algol. What makes normal order reduction especially attractive is the following theorem:

   Theorem 2 (Church-Rosser Theorem II) *If* $e_0 \stackrel{*}{\Longrightarrow} e_1$ *and* e_1 *is in normal form, then there exists a normal order reduction form* e_0 *to* e_1.

2. **Applicative Order Reduction** is the sequential reduction where the leftmost innermost reducible expression is always reduced first.

   Applicative order reduction corresponds to calling a function by value: First we evaluate the arguments of the function and then we apply the function on them.

   This form of reduction seems more useful in parallel computations, because referential transparency guarantees that the value of the arguments will be the same irrespectively of evaluation order. In this way we get at least the parallelism inherent in the parallel evaluation of the function arguments without need to apply fancy graph reduction techniques. Unfortunately, although this form of reduction seems attractive for parallel programming, no theorem analogous to 2, has been proved for applicative reduction! On the contrary there are situations, where a normal form exists and applicative order reduction fails to find it.

   Example 3 Consider the reductions of the λ-expression:

   $$(\lambda x.y)((\lambda x.x\ \ x)(\lambda x.x\ \ x))$$

   using applicative and normal order:

   - Applicative order reduction

   $$(\lambda x.y)((\lambda x.x\ \ x)(\lambda x.x\ \ x)) \stackrel{\beta-reduction(2.5)}{\Longrightarrow}$$
   $$(\lambda x.y)([(\lambda x.x\ \ x)/x](x\ \ x)) \stackrel{substitution(2.2)}{\Longrightarrow}$$
   $$(\lambda x.y)([(\lambda x.x\ \ x/x]x\ \ ([(\lambda x.x\ \ x/x]x) \stackrel{substitution(2.1)}{\Longrightarrow}$$
   $$(\lambda x.y)((\lambda x.x\ \ x)(\lambda x.x\ \ x)) \stackrel{\beta-reduction(2.5)}{\Longrightarrow} \ldots$$

- Normal order reduction

$$(\lambda x.y)((\lambda x.x\ \ x)(\lambda x.x\ \ x)) \overset{\beta-reduction(2.5)}{\Longrightarrow}$$

$$[(\lambda x.x\ \ x/x]y \overset{substitution(2.1)}{\Longrightarrow} y$$

The difference between these two computations is that that normal order reduction reduces expressions "by need" In this way infinite data structures and objects are elegant and conceptually simple to deal with. Evaluation by need is also faster because only those expressions that are needed to derive the result are evaluated. The function of the example has an argument whose computation is not terminating. However the value of the function does not depend on that argument. Normal order reduction realizes that, while applicative order does not.

Although in the above example, normal order reduction did less work than applicative order, this is not always the case. Applicative order reduction evaluates a function argument only once, so if the value of the argument is used multiple times in the function body then we save work. Normal order reduction would reevaluate it as many times as it appears in the function body. Therefore normal order reduction is faster, if we use the value of the arguments just once or not at all, while applicative order reduction is better if we use heavily the value of arguments.

If a function is *strict* in its arguments (ie. it does not terminate if the computation of any of its arguments does not terminate) then we can evaluate the function arguments before we evaluate the function without hurting the program semantics. Note that this is the opposite of evaluating "by need", but remember we are interested in parallel computation and functional languages have "referential transparency", so we may evaluate an argument once and and use it in many places.

Combinators and Recursion in λ-Calculus

Combinators were introduced in combinatorial calculus as a form of expressing all functions in terms of a basic set of functions. It has been proved that any function can be expressed via the primitive functions S, K, I defined below:

$$\mathrm{I}\ x = x \tag{2.7}$$

$$\mathrm{K}\ y\ x = y \tag{2.8}$$

$$\mathrm{S}\ f\ y\ x = f\ x\ (g\ x) \tag{2.9}$$

It is possible to remove all bound variables (all but the input variables in a functional program) from a λ-expression (functional program) using these functions:

$$\lambda x.x = \mathbf{I} \tag{2.10}$$

$$\lambda x.y = \mathbf{K}\ y \quad if\ x \neq y \tag{2.11}$$

$$\lambda x.(e_1 e_2) = \mathbf{S}\ (\lambda x.e_1)(\lambda x.e_2) \tag{2.12}$$

The power of λ-Calculus is partly due to its ability to express recursion. A function can be defined as recursive using the *Y combinator* primitive function:

$$Y = \lambda f.(\lambda x.f(x\ \ x))(\lambda x.f(x\ \ x))$$

The Y combinator has the property that for any λ-expression e:

$$(Y\ \ e) = e(Y\ \ e)$$

Example 4 Consider the following recursive definition of the factorial:

```
factorial = λ n. (if (n = 0) then 1
                             else (n * factorial(n-1)))
```

It can be written non-recursively using the Y combinator function as follows:

```
factorial = Y ( λ factorial.λ n.
                          (if (n = 0) then 1
                                      else (n * factorial(n   1))))
```

A powerful characteristic of normal order reduction is its ability to evaluate recursive function definitions. Applicative order would fell into an infinite loop, in an attempt to evaluate the recursive argument for ever.

Combinators play an important role in the implementation of functional languages, because they decompose nicely the program into components for execution. Hughes[Abramsky and Hankin, 1987] developed the idea of **supercombinators** which are a program derivable set of combinators, which make sure that no function argument is evaluated twice. Hudac and Goldberg[Hudac and Goldberg, 1985] carried the idea further and introduced the notion of the serial combinators, an extension of supercombinators which have no concurrent substructure. We will examine their technique in section 2.3.2.

Extensions to Pure λ-Calculus

Many extensions have been proposed to pure λ-Calculus to enhance its ability to express the semantics of a useful programming language. Studying them one can see λ-Calculus develop next to functional programming languages.

There is *recursive λ-Calculus with constants* which introduces the syntactic category of declarations and adds rules to transform them into pure λ-Caluclus.

Later, *typed recursive λ-Caluclus with constants* was developed. It introduces a set of typed fixed point operators for every type, In this calculus, *polymorphic functions*(ie. functions that accept arguments of different types at any position) appear and *type inference* is used to infer the type of the argument. This calculus although very attractive for practical use, has a serious drawback: Type checking is undecidable in it.

However, types were meant to be introduced in λ-Calculus. Hindley and Milner discovered a restricted polymorphic type system for which type inference is decidable. The limitation imposed to functions by the Hindley-Minler type system is that a function cannot be instantiated in two different ways within the same scope. This limitation may reject valid programs: If a function is passed as an argument, then it cannot be instantiated in two different ways inside the scope of the call.

2.2 Characteristics of functional languages

Modern functional languages[Glaser *et al.*, 1984; Bird and Wadler, 1988; Peyton-Jones, 1987] have developed a number of interesting characteristics in an attempt to combine efficiency with mathematical elegance. The efficient support and exploration of these characteristics is an area of active research:

Higher Order Functions

In functional languages computation is carried out entirely through the computation of expressions. Higher order functions reflect the ability to treat functions as first class objects: for example a function may be applied to a function and return a function as the result.

The abstraction behind this uniform use of functions and names, is that functions and names are objects alike and they should be treated as such.

Some imperative languages display similar behavior. For example a C subroutine may return a function via pointer indirection. What is missing is the concept of functions as abstractions over values: a function is considered as a

pointer to a memory block. The functional abstraction is more concise, clean, and not storage oriented which makes reasoning about issues like parallelism more straightforward.

Lazy evaluation

Lazy evaluation is an implementation of normal order reduction in which recomputation is avoided.

As we pointed out in section 2.1 normal order reduction is very attractive because it can define recursive functions via Y combinators, it evaluates arguments "by need" and most importantly if the expression has a normal form it will find it. The inherent inefficiency involved in evaluating a function argument as many times as it is used may be avoided by checking the expression to be evaluated against those that have already been evaluated.

Lazy evaluation helps the programmer reason about infinite sets like the integers, and apply functions on them, without requiring the evaluation of the whole set(which would not terminate). Actually, when we refer to lazy data structures we mean data structures whose elements are evaluated "by need" or "on demand". Infinite data structures (streams) are sometimes used in functional languages to express nondeterministic events in a deterministic way (i.e. future events or the set of continuations of a function).

A sometimes ignored advantage of lazy evaluation is that the programmer does not need to specify alternative ways of computation if he wants to optimize his program. The compiler will not evaluate anything that is unnecessary in obtaining the result. For example, if a function argument is not used in the function body then it will not be evaluated.

Lazy evaluation in functional languages is a relatively new feature. Early functional languages such as pure Lisp, FP, ML, and Hope as well as dataflow languages like Lucid[Wadge and Ashcroft, 1985] use applicative order semantics, because they are easier to implement. There is a version of ML, LML which has adopted lazy evaluation. Haskel is another recent example of a lazy language. The implementation of lazy evaluation requires lazy graph reduction mechanisms and its efficient implementation in traditional computers is an active area of research.

Pattern Matching

Pattern matching is an elegant way to define a function via multiple equations which apply under different conditions. It corresponds to a case statement in a

conventional language:

CASE *exp*
$pat_1 \rightarrow ee_1$
$pat_2 \rightarrow e_2$
.....
$pat_n \rightarrow e_n$
END of CASE

Patterns designate different ways one might use the function, or different types of arguments one might use. Syntactically the patterns may appear as alternative definitions of the same function for different types of arguments.

If the patterns are forced to be mutually exclusive, then the options can be evaluated in parallel, without overriding the notion of determinism. ML uses pattern matching in function definition, however it is not an error if multiple patterns match *exp* because the $CASE$ statement is evaluated sequentially. If the compiler decided to explore this form of parallelism in a language like ML, then priorities should be attached to the alternative definitions so that the highest priority pattern win in case of a conflict.

Pattern matching in its own right is useful in making the programs readable and understandable to the programmer. A λ-expression or a lisp s-expression is not the most readable form for a program: in some cases it has a very deep nesting and you cannot understand easily what the program means.

Strictness

Definition 3 *A function is strict on its i_{th} argument x_i if whenever the computation of x_i does not terminate, the function computation does not terminate as well.*

Suppose we have to compute:

$$f(e_1, \cdots, e_{i-1}, e_i, e_{i+1}, \cdots, e_N)$$

and f is strict on its i_{th} argument, then the evaluation of e_i can be executed before or concurrently with the evaluation of f, retaining the guarantee that the normal form if existent will be found.

In some functional languages, the user may annotates functions as strict on one or more of its arguments. In this way the user is able to supply the compiler with information that it might not be able to acquire, which if correct will lead to more efficient execution. The compiler will try to verify the user information

using abstract interpretation techniques[Abramsky and Hankin, 1987] and if it does not locate any indications to the contrary, it will accept in as correct.

Other functional languages are strict by definition: If the evaluation of a function argument does not terminate then the function application does not terminate as well. Languages that have this property are those with applicative semantics (i.e. use applicative order or "call by value" in function application like FP[Backus, 1978]).

Strong typing

Early functional languages were interpreted and untyped. Modern functional languages are strongly typed. This means that no syntax errors will appear after the functional program has been successfully compiled.

Strong typing assumes that the names in the program have been declared via special statements and/or that the compiler can *infer* the type of the names statically at compile time. The favorite type system for this new generation is the Hindley-Milner type system, which gives an expression the most general type that does not violate the semantics of the type.

Strong typing, in the way it is supported by ML and Haskel, reduces the number of run time errors because all arguments are syntactically tested. The execution time of the programs is also reduced, because the number of run time checks is reduced.

Polymorphism

Many functional languages support polymorphic functions. There are two types of polymorphism[Harland, 1984] that may be exhibited by functions:

- *ad-hoc polymorphism.*

  A function exhibits ad-hoc polymorphism if its behavior depends on the type of its arguments. For this reason ad-hoc polymorphism is sometimes called *overloading.* An example of such a function is the function '+' as we use it in imperative languages. The behavior of the function is different depending on whether the arguments are floating point numbers or integers; the function will check the arguments if they are both integers it will perform an integer addition, if they are both floating point it will execute a floating point addition, otherwise it will do the necessary conversion of the integer argument and it will execute a floating point addition. The function name ('+') is a syntactic construct to overload a notion with more than one meanings.

- *parametric polymorphism.*

  A function with parametric polymorphism behaves in the same way irrespectively of the type of the arguments. An example of this kind of polymorphism is the function "map" in lisp. "Map" applies its first argument to the elements of the second which is a list. This works no matter what the type of the function or the type of the list is.

  The Hindley-Milner type system provides this type of polymorphism and two of the most elaborate functional languages (ML,Haskel) are based on it.

2.3 Compilation of functional languages

2.3.1 Phases of the compiler

The compilation of functional programs is not fundamentally different from that of imperative programs. The main compilation stages are the same:

- *Lexical analysis,*
- *Parsing,*
- *Optimization,*
- *Code generation.*

Although optimization is usually considered as a separate phase following parsing, optimization is frequently split into the different phases of the compilation to gain advantage of the information available in each stage.

A functional language compiler has phases which are nonexistent and/or have different flavor in traditional compilers. Here we describe them giving emphasis to their importance in exploiting program parallelism:

- *Type checking,*
- *Optimization,*
- *Removal of pattern matching,*
- *Variable abstraction or λ-lifting,*
- *Strictness analysis,*

- *Boxing analysis,*
- *Storage class analysis,*
- *Code Generation,*

Type checking

Type checking is a part of every compiler for a typed language. In this phase syntactic and type errors are detected and reported to the programmer. A type checking algorithm will fail if a run time error occurs due to a type mismatch.

Most modern functional languages have a polymorphic type system based on Hindley-Milner semantics. An object is assigned the most general type expression under the condition that its code acts identically on all instances of that type expression. This is where parametric polymorphism goes it. If for example, the function can be applied to both arrays of integers and arrays of floating point numbers, the type system will adopt a type "array" for the function argument.

The programmer has the choice of either using the type declarations to denote the type of a function and its arguments, or let the compiler infer the relevant types according to the rule of the previous paragraph.

Although type checking is not directly related to parallelism, it is as important as the syntax in the program because it helps the programmer run correct programs, by detecting at compile time the frequent syntax errors in the program.

Optimization

Here we refer mostly to parse phase optimizations like common subexpression elimination, expression evaluation, inlining, data transformations, and unfolding.

The lack of side-effects in functional languages makes common subexpression elimination and expression evaluation optimization applicable without the restrictions that apply to imperative languages.

All function applications on arguments available at compile time can be evaluated at compile time and therefore reduce runtime execution.

Inlining is an attractive form of optimization because it saves the overhead function call/return. Unlike most imperative languages where function calls have relatively coarse granularity, in functional languages the function call is the main abstraction and thus has fine granularity. Inline substitution is justified by the

potential decrease in execution time, however the frequency of function calls in a functional program results in a substantial increase of program size.

Unfolding recursion, transforming data types, allocating memory in chunks for lazy data structures all result in a faster program of much larger size.

Pattern Matching

Pattern matching cures the problem of very deep s-expressions resulting in more readable programs. It is also a source of parallelism. The determinism of functional programs implies that only one of the alternative function definitions may apply at any use of the function name. Multiple definitions can be checked out in parallel, and if one succeeds the remaining processes are killed. In that case the runtime system, maybe a task manager has to be able to deal with the so called "irrelevant tasks". We call irrelevant, the possibly non-terminating tasks whose execution has begun and whose value is not needed. Such tasks appear when we evaluate both branches of an "if" predicate concurrently to the "if condition", or more generally when a non-needed function argument is evaluated concurrently to the function body, simulating "call by value".

There is a hidden danger in trying to exploit the parallelism in pattern matching. Usually the definition of a recursive function(quite common in functional programs!) has two parts, a recursive definition and an exit condition. For example, the fibonacci sequence may be defined as:

$$
\begin{aligned}
fibonacci(0) &= 1 \\
fibonacci(1) &= 1 \\
fibonacci(n) &= if \;\; n > 1 \;\; fibonacci(n-1) \;\; + \;\; fibonacci(n-2) \\
&\quad\;\; else \;\; \perp
\end{aligned}
$$

If we generate a separate process to check each of the possibilities without considering the size of the job it is quite possible that the resulting parallel program will be much slower from the serial one! The cost of creating a process and destroying it in the end is quite probably greater than the speedup from checking the possibilities in parallel.

For this reason most functional compilers today substitute pattern matching in the program by conditionals and then they use some global reduction technique to partition the program into concurrent processes.

Variable Abstraction or λ-lifting

After the compiler removes all pattern matching from the program, lexical scoping has to be removed. Lexical scoping is usually introduced by the "where" clauses:

```
f(x) = ···
       where g(y z) = ···
```

The variables y and z introduced by function g are neither local not global to the program, they are formals of g. A simplistic solution would be to just take g's definition and place it in the same scope as f, changing its name to g' to avoid any conflict with another function named g.

However things are not as simple. g may refer to variables local to f because it is defined inside the scope of f. This can be solved by passing explicitly as parameters to g whatever names it is using. More complications arise from the existence of mutually recursive functions and the general form of the λ-lifting algorithm is fairly complicated [Hudac and Goldberg, 1985]

λ-lifting is the core of the *supercombinators analysis* which breaks the program into basic functions so that no recomputations arise in the execution of the program(i.e. you do not evaluate the same function application twice). This approach is correct because due to referential transparency the value of an expression depends only on the syntactic context.

Not all compilers for functional languages apply λ-lifting. The alternative approach requires to carry around the environment comprised of the names defined in the current scope.

Strictness analysis

Strictness analysis is used to find function definitions and applications where applicative order does not change the semantics of normal order evaluation. A functional compiler should gather information about the strictness of the program functions so as to be able to exploit as much parallelism in the program as possible. If the compiler realizes that a function is strict, or the function has been annotated as such, then applicative order reduction can be applied without fear.

The techniques used to apply strictness analysis to a program are based on abstract interpretation [Abramsky and Hankin, 1987]. Each function f definition and application is syntactically analyzed and based on domain and range

information derived from its type we test for each argument if:

$$f \perp = \perp$$

Based on this analysis the "function-argument" pairs are marked as strict, non-strict, or don't know.

Unfortunately due to run time dependencies, the compiler cannot always decide the truth of the statement. In that case, the function is assumed to be *nonstrict* in its argument.

Storage class analysis

The evaluation model in conventional languages in mostly *stack based.* However the stack has an inherently centralized structure which would act as a bottleneck in the execution of a functional program.

The most significant problem is created by higher order functions which require a heap allocated data structure known as *closure* to be passed around.

A functional program generally requires more run time memory than a an equivalent one in a language of the Algol family. The compiler should treat memory carefully and classify and annotate program names according to their preferred storage class

Boxing analysis

The output of the parsing phase is a tree structured representation of the expression being evaluated. An application node has the applied function as the left subtree and the parameter expression (value of the argument) as the right subtree.

When the function call is evaluated, a pointer to the argument structure is passed to the function body. If the same argument expression is passed to multiple functions, there is no need for it to be evaluated multiple times, in the same context; it will always represent the same value. This observation improves the efficiency of the evaluation model by transforming the initial tree structure, into a directed acyclic graph structure. At this point an argument expression may have two parents(i.e. belong to different function applications). The first time we evaluate it, we overwrite the top node representing the argument expression and all subsequent uses of the argument will take it evaluated.

Following pointers is considered an expensive operation when compared with register operations. This is what boxing analysis is about: A parameter is called *boxed* if you have to follow a pointer to get its value.

Boxing analysis finds out which argument expressions will be evaluated at the time of the function application evaluation and produces code that places them in a register. The phase is not trivial because the compiler has to find out which arguments are used and create code only for those.

The resulting code is much faster than the naive one, and in combination with strictness analysis, substantial speedup is achieved.

2.3.2 Parallel graph reduction

The stack oriented execution model is not very efficient for functional languages. Researchers attempted to construct a framework where the execution of functional languages would be efficient, so they developed parallel graph reduction machines where reduction can be done quickly like ALICE [Darlinghton and Reeve, 1981], and ALFALFA [Hudac and Goldberg, 1985].

The basic idea of this approach is that while the stack is a useful abstraction for subroutine call/return in conventional languages, in functional languages most objects are heap allocated. For example, if the programmer wants to pass a function as an argument to a function he has to pass a pointer to the function.

Most importantly we want to exploit the parallelism that exists in processing function arguments concurrently. The major question here is when to create a new process to operate on a part of the program graph, and how the created processes interact, cooperate and synchronize. The approach of many researchers to this problem is transform the program into a set of combinators first and then separate the program at the right granularity.

- Express the program in terms of a *fixed set of combinators* like the well-known *I, K, S*(defined in section 2.1.1). However, simulation has shown that this is not a good idea. This method may decompose for parallel execution purely sequential programs, whose data dependences preclude any parallelism [Hudac and Goldberg, 1985].

- Express the program in program derived *supercombinators* [Hughes, 1982]. The important property of supercombinators is that no subexpression is recomputed. The problem is that the technique was made for sequential machines without any concern for parallelism. In this way some program parallelism may be lost in the process.

- *Serial combinators* [Hudac and Goldberg, 1985] were developed aiming to alleviate this problem. Their goal was:

> "to keep the environmental nature of the combinators and their usefulness in graph reduction while maximizing the granularity and ensuring that parallelism is not lost."

Indeed serial combinators is a very good method for parallel reduction, because the have no concurrent substructure and thus they guarantee that no parallelism is lost and they result in fully lazy evaluation so that no extraneous computations are performed. The problem with this method is that there are naturally parallel programs that cannot be decomposed using this method [Kelly, 1990].

In most multiprocessors the ratio of computation to communication varies roughly form 10 to 100. This means that the cost of communication is nontrivial and coarser granularity is expected to give greater speedup. The problem is aggravated because these decisions are largely architecture dependent. A granularity gives great pay off in one case may be deadly slow in another.

Generally speaking we have to consider the overhead involved in creating a new process, and having it communicate with the rest of the processes, against the speedup achievable by parallel execution. The creation of a new process involves:

1. The new process is placed in the process pool for scheduling. Its descriptor will contain a reference to the expression graph and the identifier of the process that spanned it. We assume the existence of a centralized job queue where jobs are entered by executing processors and are retrieved by idle processors.

2. The new process is moved to a remote processor and is scheduled for execution.

3. Synchronization in the program graph. If we assume that the program is executed by multiple processors, the program graph has to be protected. Two processors may request the value of an expression at the same time, evaluate it and write back the result. If no synchronization is provided this might result in inefficiency or even worse inconsistency. If another processor is evaluating an expression a given process needs, then the process should block. Its processor might pick another process task to execute. With each node that is locked we may associate a pending list, so as to be able to resume promptly whatever processes are blocked because of that node.

We refer to some of the problems we face when we decide to do parallel execution of functional programs using parallel graph reduction. The main question is the following:

> How can we decide wether the evaluation of a subexpression achieves speedup?

Here we will describe the approach taken by Hudac and Goldberg in the design of a compiler for ALFALFA [Goldbelg, 1988]. Suppose we have to compute:

$$exp_1 \;\; op \;\; exp_2$$

The serial execution cost of the above expression is :

$$C_{serial} = T(exp_1) + C(op) + T(exp_2)$$

where $T(exp_i)$ is the cost of executing exp_i $(i = 1, 2)$ and $C(op)$ is the cost of computing the result. The concurrent execution cost of the same expression is:

$$C_{parallel} = max(T(exp_1), T(exp_2)) + C(op) + Concurrency - cost$$

Clearly if $C_{serial} \leq C_{parallel}$ we do not gain anything by the concurrent execution and we slow down the whole execution by exploiting this parallelism.

The time for the evaluation of an expression is approximated using a heuristic in which expressions are weighted by the complexity of the primitive operations. The advantage of this approach is that it can be tuned to different architectures.

Keep in mind that here we are not considering the natural bounds in speedup that the nature of the program may have. For example we now that we can do sorting in O(logn) time, and that is optimal. This is a theoretical bound which we cannot improve. However the software and the hardware available may be improved so that we can get closer to that bound.

2.4 Parafunctional programming

When a functional program is executed on a multiprocessor, a set of processors cooperate decomposing and distributing the computation among themselves. This program mapping occurs at the compiler level and the programmer may choose to ignore the fact that his program is executed in a parallel processor.

Simulation results show that the techniques used by functional compilers are comparable in efficiency to those for imperative languages. On the other hand no matter how good the compiler is, it can not find out as much as the user about

the program and its behavior (in acceptable time). Here we face the traditional trade-off between having the user make the mapping decisions and having the system make them at the cost of lowering the efficiency of the system.

If we allow the programmer to specify an ideal mapping for his program on a general multiprocessor then "optimal" performance can be achieved. Such a mapping can be specified by annotations. Annotations are notes attached to expressions, giving the compiler hints on where the expression should be executed. It is important that the annotations do not alter the functional semantics of the program (i.e. the result of the program should be the same with or without annotations).

A significant advantage of the "annotation" concept is that it separates the algorithm from the execution model. This makes easier the testing of different process structures on the same program with minor code changes[Kelly, 1990] and hence program portability is made easier. Another advantage of this separation is that the programmer can debug a functional program on a uniprocessor and then he can improve its multiprocessor execution via annotations.

Paul Hudac [Hudak and Smith, 1986; Hudac, 1986] developed these ideas into a *parafunctional* methodology which allows the programmer to express explicit parallelism via annotations and applied them on the design of *ParAlfl.*

2.4.1 ParAlfl -A parafunctional programming language

ParAlfl is a block structured, lexically scoped language with lazy evaluation semantics and pattern matching.

A program in ParAlfl is a group of equations. An equation group looks like this:

$$
\begin{aligned}
\{\ & f_1(x_{11}, x_{21}, \cdots, x_{k1}) == e_1; \\
& f_2(x_{12}, x_{22}, \cdots, x_{k2}) == e_2; \\
& \quad \cdots \\
& \textit{result exp;} \\
& \quad \cdots \\
& f_n(x_{1n}, x_{2n}, \cdots, x_{kn}) == e_n; \}
\end{aligned}
$$

The equation group defines a set of local indentifiers and contains a single return statement which reflects the value it evaluates to. Identifiers within the group are evaluated "on demand". Equation groups may be nested at arbitrary depth. The order of the equations in the group is irrelevant. Pattern matching is used to make easier the definition of complex expressions.

The programmer may supply information on the desirable program mapping at different levels of detail by using two types of annotations:

- *mapped expressions.* Mapped expressions are annotations of the form *exp $on proc*, specifying that the subexpression *exp* should be executed on processor numbered by the value of *proc*. Processors are assumed to be numbered by integers and *proc* may be any integer value expression. For example consider the following annotated expression:

  $$f(x) \ \$on \ \ 0 + g(x) \ \$on \ \ 1$$

  it has the effect of computing f(x) on "proc" numbered 0 and and g(x) on processor number 1 and then adding the results on the currently executing processor.

- *eager expressions.* An expression may be specified as eager by prefixing it with the pound (#) sign. An eager subexpression may be executed concurrently with its outer scope. In other words the programmer can give strictness hints to the compiler (2.2). For example in

  $$f(\#x, y)$$

  the evaluation of x begins concurrently with the evaluation of f when the value of f is requested.

Reference to the currently executing processor is allowed by the dynamic variable *$self*. In different executions of the same program with the same arguments the value of $self may be different, so the ability to refer to $self violates referential transparency. If the use of $self were not allowed then the programmer would not be able to implement a mess, or an execution tree.

They managed to overcome this difficulty by proving the following theorem:

Theorem 3 *[Hudak and Smith, 1986]*
A ParAlfl program in which: (1) the identifier $self appears in the "proc" part of a mapped expression and (2) all "proc" expressions terminate, is functionally equivalent to the same program with all annotations removed.

According to the theorem we may use $self without changing the semantics of the program as long as we use it only in the evaluation of "proc". The theorem condition although sufficient it is not necessary for the annotated program to be functionally equivalent to the unannotated. A functional program may use $self in general expressions and still be deterministic. Unfortunately however, no tighter condition is known.

The following example illustrates the use of mapped and eager expressions in a ParAlfl program that computes the first n primes using the "sieve of Eratosthenis".

Example 5 A call to the following program returns the first n primes. The program is based on the ability to handle infinite lists. The "ints" and the "primes" are infinite lists which are evaluated as needed. The evaluation of these functions ordinary would not terminate, however since the language is lazy, only the prefix of the set requested will be evaluated.

We get the list of all primes by "filtering out" all multiples of the numbers established as primes. In line 4, we see notice the separation of a list into the head and the tail, p is the first element in the list and rest is the remaining elements. "filter" in line 5, takes a list as argument and and returns a list without the multiples of the first list element. Line 5 displays the ParAlfl conditional which is of the form

$$condition- > exp_{TRUE},\ \ exp_{FALSE}$$

and has the standard meaning:

$$if\ \ condition\ \ then\ \ exp_{TRUE}\ \ else\ \ exp_{FALSE}$$

In the specific case if n is a multiple of p, discard it and filter the rest, otherwise keep it and filter the rest.

Lines 9 and 10 define integers as an infinite sequence, while lines 10 and 11 define $prefix(n, l)$ as the list containing the n first elements of the list l.

```
(1)  { result prefix(n, primes);
(2)    primes == sift(ints);
(3)
(4)    sift(p ^ rest) ==
(5)                  { result p ^ (sift(filter(rest)) $on right($self));
(6)                    filter(n l) == n | p == 0 - >filter(l),
(7)                                                n ^ #filter(l)}
(8)
(9)    ints = numsfrom(2);
(10)   numsfrom(n) == n ^ numsfrom(n+1);
(11)   prefix(0, l) == [];
(12)   prefix(n, a l) == a ^ #prefix(n-1, l)
```

The annotations for eager expressions are really interesting. If we remove them then the program possesses no parallelism at all; each prime filters in turn the rest of the numbers and finally the n smaller numbers may be evaluated.

To allow for parallelism we annotate the expressions $prefix(n-1, l)$ on line 5, and $filter(l)$ on line 10 as *eager*. In this way the filtering occurs in a pipeline fashion each processor filtering out a different prime. This approach however brings out again the problem of "irrelevant tasks". After the first n prime numbers have been evaluated, the task manager has to go off and kill all remaining tasks.

The processor structure assumed here is that of a infinite array. This can be seen on line 4, where the expression is mapped on the right of the currently executing one ($self). This infinite sequence of processors can be mapped underneath on a mess, a ring or a hypercube. The specification of parallelism in this example is not a really tight one, the process structure could have been explicitly specified as a ring or a hypercube, even if the underlying architecture were not such.

A number of extensions to this parafunctional methodology have been proposed (see [Kelly, 1990]). They involve ways to extent the current scheme by giving the user access to information such as load balancing, memory utilization or number of available processors.

A different type of extensions involves the ability to map operating system resources like storage devices or I/O channels on a user data structure.

2.5 Case studies in functional languages

One may argue that functional languages where born with lisp and that would be close to reality. The ideas were floating around as an alternative way to think about programming and Lisp was born.

Afterwards many functional programming languages appeared with more or less significance, however each and every one of them contributed to the significance of the field. There was APL[Polvika and Pakin, 1975] the first language to use functional notation. Then more came: HOPE[Field and Harrison, 1988], Miranda[Turner, 1985], ML[Milner, 1984; Appel and MacQueen, 1990], Lucid[Wadge and Ashcroft, 1985], VAL[McGraw, 1982], Haskell[Hudac and Wadler, 1988]. Some of them run today on parallel computers, some of them are still maturing little by little.

2.5.1 Pure Lisp

Lisp has a pure functional subset. McCarthy[McCarthy, 1960] conceived Lisp out of the λ-Calculus context as a symbol manipulation language for AI. It supports all the basic characteristics of a functional language but none of the fancy ones. It is well known for its elegance and power(ie. one can write a lisp interpreter in just 30 lines of lisp code).

While functional languages are used to exploit multiprocessors organized in a hypercube [Hudac and Goldberg, 1985; Arvind and Ekanadham, 1988], a variant of Common Lisp [Steele, 1984], CmLisp was used to program the Connection Machine[Hillis, 1985]. The Connections Machine is a highly interconnected SIMD

(SIMD = Single Instruction, Multiple Data) network of 1000-10000 processors (aiming for more in the future).

All the processing elements in the Connections Machine execute the same instruction at the same time on the local data. Actually there are switches associated to processing elements, so that a processing element may decide it does not want to operate on some cycle. CmLisp has a data structure and parallel operations on it to exploit data parallelism. Although the language itself is an explicitly parallel language it is interesting, because it exploits *data parallelism* in a functional language. This is accomplished by incorporating parallel data structures xectors, operations and semantics on them.

Traditionally the type of parallelism exploited in functional languages is *argument parallelism*(Argument parallelism results from evaluating the function arguments in parallel).

2.5.2 ML, SML

ML(Meta Language) is a general purpose language with a very powerful functional subset. It was developed in the mid '70s as a language for building proofs. In the early 80's Standard ML (SML) [Milner, 1984; Appel and MacQueen, 1990] was developed from ML with extensions from HOPE. It is important because it was one of the first full programming languages to be based on well defined theoretical foundations.

ML is a strongly typed language, every program variable or function has a type. However type declaration is not mandatory, because the language uses type inference. ML also supports function and type polymorphism. The programmer is able define new types in ML.

One of the expressibility problems with functional languages is that they cannot directly support abstract data types. ML has gone around the problem by the introduction of *modules* to group functions by meaning.

ML is not a pure functional language. It even has pointers references. However it is has well-defined semantics and it encourages the programmer to write side effect free programs with strong functional flavor.

What makes ML so attractive and innovative is that it supports the notion of *environment* as a first class object. This is a very powerful idea, which in effect says that the evaluation of an expression in a given environment, returns a new environment.

Exceptions are also first class objects in ML. An exception does nothing more than move you from one environment to another. This is a very important idea because it gives the right flavor to bindings, without using to the notion of

storage.

To my knowledge, no attempt has been made to built and evaluate SML in a parallel environment. On the other hand it stands somewhere between the functional and imperative paradigm and combines the advantages of both. The development of a parallel ML would be a very interesting task.

Example 6 To get a blend of ML, its conciseness and elegance here is the famous *Union-Find* algorithm written in SML:

ALGORITHM 1.1 Union-Find in SML

```
(1)  datatype 'a setelem
(2)       = NILSET
(3)       | ELEM of 'a * 'a setelem ref * int ref
(4)  exception SETINFO

(5)  fun new_setelem c = ELEM(x, ref nilset, ref 1)

(6)  fun set_union(NILSET, f) = f
(7)       |set_union(e, NILSET) = e
(8)       set_union(e an ELEM(_, e_next, e_size),
(9)                 f as ELEM(_, f_next, f_size)) =
(10)           if !e_size < !f_size
(11)           then (f_size := !e_size + !f_size; e_next := f)
(12)           else (e_size := !e_size + !f_size; e_next := e)

(13) fun find NILSET = NILSET
(14)      |find (e as ELEM(_, ref NILSET, _)) = e
(15)      find (ELEM(_, f, _)) = let g = find !f in (f := g; g) end
(16)
(17) fun setinfo NILSET = raise SETINFO
(18)      | setinfo e = let ELEM(x, _, _) = find e in x end
```

On lines 1 to 3, the type "a" is defined. "a" is either the empty set, or a 3-tuple of an set element(the top element), a pointer to a set element(the next set element) and a pointer to an integer(the cardinality of the set).

On line 4, the exception SETINFO is declared. SETINFO is raised when the function "setinfo" defined on lines 17-18 is called with argument an empty set. "set info" is the function responsible for I/O, using it we can derive information on a certain element.

Function "new_setelem" on line 5, creates a new set and makes c its top element, the next element NILSET, and the cardinality 1.

The core functions of the algorithm are "set_union" and "find" defined on lines 6-12 and 13-15 respectively. Notice the use of pattern matching in the function definition. The symbol | is used to denote alternative function definitions. Lines 6,7 says that the

set union of a set s with the empty set (NILSET) is the s independently to argument order. Lines 10-12 is the general case when both sets are nonempty. It checks which set has more elements and it sets the top element in the top set to it and it makes the cardinality of the set equal to the sum of the cardinalities.

2.5.3 Id Nouveu

Id Nouveu[Arvind and Ekanadham, 1988] is a functional language for scientific programming that allows debugging at the highest possible level. At the same time it aids the generation of efficient code for parallel machines.

Its implementors contrast the declarative style of programming of good old Fortran with the addition of annotations to that of functional programming and of Id-Nouveu in particular. They support that that Fortran is no good because it does not have sufficient run time support and lacks efficient subroutine return semantics.

Functional programming has of determinism and independence of evaluation order (i.e. Church-Rosser property), and higher order functions. It lacks the traditional storage facilities of imperative languages to implement arrays, and matrices. It also lacks nondeterminism.

They address these problems using I-structures[Arvind *et al.*, 1989; Arvind, 1982; Arvind and Ekanadham, 1988]. An I-structure is the single assignment of the traditional array allocated dynamically. One cannot read an entry of the array that has not been assigned and once an array entry is assigned a value you cannot change it. I-structures are implemented on tagged data flow architectures.

2.6 Advantages of Functional programming

Functional languages lack assignment. The lack of assignment does not prevent the construction of elaborate data structures, it just requires the dependencies between operations applied to data be made explicit.

This is a major advantage for parallel applications. The programmer does not have to deal explicitly in his program the issues of communication and synchronization, the underlying system will do what it takes. No worries about who has locked what, how did it get the lock, and so on. The program may be debugged in a sequential computer and the compiler is responsible for generating efficient code for the multiprocessor.

Functional languages have referential transparency. Any functional program is deterministic by nature, each expression has one value that depends only on

the arguments of the expression. This property makes common subexpression elimination trivial. No need to worry about the life span of the variables, in the expression. The application of parallel graph reduction techniques very straight forward.

Functional languages are based on one primitive: *functions.* This results in a small elegant language with clear semantics, easy to reason formally about.

Polymorphism and strong typing in functional languages make program development easier and faster and programs compact and more concise. The programmer does not need to write different functions for different data types, in this way he writes less code and he makes less mistakes. Most programmer errors are type errors; by making the language strongly typed the programming task is substantially reduced.

Most importantly however, the programmer may ignore that his program will run on a parallel machine, no complicated state tracking, no locking problems, everything is handled underneath. The advantage of the approach to program development is amazing.

Functional languages have clear operational semantics: A function is applied to a set of arguments and a value is produced. These semantics make it the closest to the imperative paradigm in the family of decalrative languages. This in my opinion is the greatest advantage of all: the world understands their semantics fairly well and good front ends can be written that make the switch from a language like C or Pascal to the functional paradigm easy.

It is that functional languages introduce a new abstract programming philosophy:

Do not use variables, do not program thinking about the storage...

Parallel graph reduction techniques, which are fairly explored, provide a very suitable execution model. Substantial research is under way in the area and there are functional languages running on hypercube architectures with apparent speedup when compared with sequential ones.

A final advantage of functional programming is the ability to extend the general model using annotations and give the user the ability to control the configuration of the process network his program is going to operate on (parafunctional programming). This is a great asset for a programming language: If the user believes that he cares enough to do the job, the language gives him the tools and the expressive ability to do it. If on the other hand he is satisfied with what the performance offered by the compiler, he is done at the point he has finished debugging the sequential code.

2.7 Disadvantages of Functional programming

Unfortunately, functional languages are not without disadvantages. Most of the disadvantages are related to their implementation. The current implementations of functional languages are rather slow. In addition to that some of the ideas on which the language is based like dynamic memory allocation, lazy semantics and single assignment and high order functions do not have fast implementations.

Single assignment creates another significant implementation problem. Under single assignment:

- if the program tries to read an unbound name, then it is either a run time error or program control is blocked till the name is initialized.
- if the program tries to write a bound name then a run time error is triggered.

This is a reasonable implementation for single assignment, but very expensive to implement. Arvind [Arvind, 1982] implemented a schema similar to the above on a dataflow (tagged) architecture. The problem is called the "update" problem and researchers try to find efficient ways to reuse the memory in structural ways rather than doing the standard garbage collection. Arvind, in [Arvind and Ekanadham, 1988] describes different ways to reuse an array that undergoes transformation in numerical analysis applications.

Dynamic memory allocation and garbage collection are inherently slow. To limit the effects of these issues excessive memory is used. There is need for new techniques built to suit the behavior of functional language. The imperative model is not adequate. For example, imperative programs usually execute portions of code that are close (i.e. locality). Most techniques for memory management are based on it. This is not as profound in functional programs, where most operations are one form of reduction or another. The order in which program execution(evaluation) will proceed depends more on the semantics that on the syntax of the program. For example, if a function is strict on one of its arguments the evaluation of the argument will happen concurrently to the evaluation of the function.

Lazy semantics require dynamic memory allocation as it is not possible to allocate statically a list that the program will traverse lazily. If now the system allocates memory for the list on demand, what is the the right granularity of allocation? If the allocated size is small then you loose in time, if it is big then you loose in space!

Attempts have been made to built multiprocessors whose operation is based on graph reduction [Darlinghton and Reeve, 1981] but the fact is that parallelism

requires a high level language that can run on different type of machines without paying significant performance penalty due to porting.

A more significant problem with functional languages is that they are a little awkward in expressing abstractions. Abstract date types is a peculiar object in the context of the functional paradigm. Recent functional languages employ lexical copying and try to merge save the case using signatures, but generally the problem is not adequately handled.

As far as parallelism is concerned functional languages are able of providing naturally medium to coarse grain parallelism as the naturally supported type of parallelism is argument parallelism, introduced by strictness analysis. If the compiler cannot exploit that coarseness, possibly because the size of the generated tasks is not large enough to pay off for the cost of managing them, then the program will be executed sequentially. This is where the importance of the process specification language[Kelly, 1990] or parafunctional programming is. Changing the configuration on which the program will be executed, will improve the performance with hopefully minor code changes.

2.8 Conclusions

The main advantage of the functional paradigm is that it stands between the imperative paradigm and a purely declarative one.

Modern functional languages have some imperative characteristics which make them more useful to programmers, and facilitate more efficient language implementations.

They have been used for parallel scientific programming and techniques have been developed that expose the parallelism in the program and result in higher efficiency. If an program can be written and tested that can find the parallelism in matrix multiplication, the programmer can take advantage of it and reduce the development cycle of his program.

Functional languages do not provide for large grains of parallelism and that is partly due to its underlying model of functions. Functions have nice operational semantics (application), but they are not adequately abstract. Top down design issues are very relevant important and ML and Haskel have made a large step towards that direction.

Closing, as far as automatically detectable parallelism is concerned, functional programs are best for either tightly-coupled multiprocessors or a hypercubes. In tightly coupled multiprocessors interprocess communication is very fast and the cost of creating and scheduling multiple processes is effectively

reduced. In loosely coupled multiprocessors, the relatively fine granularity of the parallel tasks combined with the high communication cost reduces speedup significantly.

Chapter 3

Logic Programming Languages

3.1 Introduction

The abstract model of logic programming languages is based on First Order Predicate Calculus. A logic program is composed out of facts and rules. The output of the program is "proved" or "deduced" from the program and the input.

The logic programming paradigm has attracted a lot of attention during the last decade partly due to the Japanese project on Fifth generation parallel computers[ICOT, 1988]. The project involves the development of parallel computers, in which the operating system and support software is written in a parallel version of Prolog. In the early years of computers and later on, operating systems were written in assembly, and were very tightly attached to the machine. Nowadays the 90% of the code in the most popular operating system, Unix [Ritchie and Thompson, 1974] is written in C, a high level language. However even in this setting, writing a program in language generally accepted as slow, was a breakthrough. No matter how high level C may be, it gives you the ability to access and manipulate the machine state directly.

Prolog is the most widely known logic programming language[Clocksin and Melish, 1981]. It is based on Horn Clauses and it was initially built as a vehicle for symbolic computation and theorem proving.

In what follows we will first describe the abstract model of logic programming and then we will describe the basic aspects logic programming languages, the problems met in the design of parallel interpreters for logic programming

languages and finally the advantages and disadvantages of logic programming with respect to parallelism.

3.2 The logic programming paradigm

To get into the spirit of logic programming notation here are some basic definitions from [Shapiro, 1989]:

Definition 1 Term *is a variable or a function symbol of arity* $n \geq 0$ *applied to* n *terms.*

Definition 2 Atom *is a formula of the form* $P(T_1, \cdots, T_n)$ *where* P *is a predicate of arity* n *and* $T_1, \cdots, T_n$ *are terms.*

Definition 3 *A* **logic program** *is a finite set of definite clauses:*

$$A_i \longleftarrow B_1, B_2, \cdots, B_n \quad n \geq 0$$

where A_i *is an atom called the* head *of the clause and* $B_1, \cdots, B_n$ *is a sequence of atoms, called* body *of the clause. Intuitively* $B_1, \cdots, B_n$ *represent the subgoals that need to be proven true before we conclude that* A_i *is true. If* $n = 0$ *then the clause is called a* fact, *and it holds unconditionally.*

Definition 4 Goal *is a sequence of atoms* $A_1, \cdots, A_n$. *A goal is* atomic *(if* $n = 1$*), or* conjunctive *(if* $n > 1$*).*

Definition 5 *The* **vocabulary** *of a logic program* P *is the set of predicates and function symbols that occur in clauses of* P.

An important feature of logic programming is the clear distinction between declarative and operational semantics[Kacsuk, 1990; Shapiro, 1989]. Declaratively each clause in a logic program is read as a universally quantified implication. If $X_1, ..Xn$ are the variables in the clause $A \leftarrow B_1, \cdots, B_k$ then the declarative meaning of the clause is:

$$\forall X_1 \cdots X_n, \quad A \ \ only \ \ if \ \ B_1 \wedge B_2 \wedge \cdots \wedge B_n$$

The logic program itself is read as the conjunction of the universal implications that correspond to its clauses.

Operationally logic programs can be thought of as an abstract computational model, like the Turing Machines. A computation in this model is a goal driven

reduction from the clauses of the program. The output of the program is reduced to the facts of the program. The "state", $< G; \theta >$ of the computation consists of a goal G and a variable substitution θ. The initial state of the computation $< G; \epsilon >$, consists of an initial goal to be proven and the empty substitution ϵ. The computation progresses nondeterministically from state to state by the rules *reduce* and *fail*. When we reduce, we intuitively substitute a composite goal, with a set of simpler ones, unifying the head of a close with the goal we are considering and composing a new substitution from the current one and the one induced by the unification. A fail rule is bad news, it means that the computation can proceed from the current point because the current goal cannot be satisfied in the current data base.

At each state the goal represents a statement whose proof will establish the initial goal. The substitution represents the values computed so far for the variables in the computation. The computation ends in a state whose goal is either TRUE or FAIL. If the final goal is TRUE then the substitution θ restricted to the variables in the initial goal is called the "answer substitution". The initial goal instantiated by the answer substitution is a logical consequence of the program. The answer substitution essentially says that the input is consistent with the program and here is the satisfying substitution.

To define formally the state transitions we need to consider a few definitions first:

Definition 6 *A* **unifier** *of two terms T_1 and T_2 is a substitution θ if the terms we obtain by substituting all variables X in T_1 and T_2 by $\theta(X)$ are the same.*

Example 1 Consider the terms $T_1 = f(X, a, Y, Z)$ and $T_2 = f(b, W, Y, Y)$.
A unifier τ for these terms is $\{(X, b), (W, a), (Y, c), (Z, c)\}$, and the term we obtain if we apply this substitution is: $T_{1,2} = f(b, a, c, c)$

Definition 7 *A substitution θ is a* **most general unifier(msg)** *of two terms T_1 and T_2 if it is a unifier of T_1 and T_2 and any other unifier θ' for them can be obtained as the composition of θ and some other substitution σ, namely $\theta' = \theta \mathbf{o} \sigma$.*

Example 2 Consider the unifier of example 1, $\tau = \{(X, b), (W, a), (Y, c), (Z, c)\}$ for $T_1 = f(X, a, Y, Z)$ and $T_2 = f(b, W, Y, Y)$. This is not a most general unifier. The most general unifier θ for T_1 and T_2 is $\{(X, b), (W, a), (Y, Z)\}$ and $T_{1,2} = f(b, a, Y, Y)$. Furthermore $\tau = \theta o \sigma$ where σ is the substitution $\{(Y, c)\}$.

The procedure of finding the most general unifier of two terms is called unification.

The modeling of the state transitions is a logic program is done using the notion of a transition system[Shapiro, 1989].

Definition 8 *A* **transition system** *for a logic program P consists of a set of states of the form $< G; \theta >$ and a set of transitions. A transition is a function from states to set of states.*

The two most important transition rules are reduce and fail:
◇ *reduce:*

$$< A_1, \cdots, A_i, \cdots, A_n; \theta > \xrightarrow{reduce} < A_1, \cdots, B_1, \cdots, B_k, \cdots A_n); \theta o \theta' >$$

if $mgu(A_i, A) = \theta'$ for some clause $A \Longleftarrow B_1, \cdots, B_k$ in the program.
◇ *fail:*

$$< A_1, \cdots, A_i, \cdots, A_n; \theta > \xrightarrow{fail} < fail; \theta >$$

if for some i and for clause $A \longleftarrow B_1, \cdots, B_k$, $mgu(A_i, A) = fail$.

Computation in the abstract logic programming model is by virtue nondeterministic. There are two types of nondeterminism in the reduce transition: *And-nondeterminism* and *Or-Nondeterminism.* And-nondeterminism corresponds to selecting which goal clause to reduce first and Or-nondeterminism corresponds to selecting which clause to reduce it with.

These types of nondeterminism give raise to two distinct types of parallelism:

- *And-parallelism* which exploits the ability to satisfy two subgoals of a given goal in parallel and
- *Or-parallelism* which exploits the ability to satisfy a goal in multiple ways at the same time.

Formally a computation of a program P on a goal G is a finite or infinite sequence of states: $c = S_1, S_2, \cdots$ such that the following conditions hold:

- $S_1 =< G; \epsilon >$,
- for each k, $S_{k+1} \in t(S_k)$ where t is a transition system for P,
- c is finite and of length k only if S_k is a terminal state.

The transition system for logic programs realizes in effect a proof procedure for logic programs. Each reduction is an application of an inference rule called SLD-resolution [Hill, 1974; Clark, 1979; Lloyd, 1987]. SLD-resolution and therefore the transition system has soundness and completeness results that link the logical point of view to the operational one.

Theorem 1 Soundness and Completeness of SLD-resolution[Hill, 1974; Clark, 1979; Lloyd, 1987]

Let P be a program and A be an atom.

(1) Soundness: If P has a computation on the initial goal A with answer substitution θ, $(\forall)A\theta$ is a logical consequence of $(\forall)P$

(2) Completeness: IF $(\forall)A'$ is a logical consequence of $(\forall)P$, where A' is an instance of the atom A, then there is a computation of P on the initial goal A with answer substitution θ_p such that A' is an instance of $A\theta$.

3.2.1 Prolog

Prolog[Kowalski, 1974; Roussel, 1975; Clocksin and Melish, 1981] is the most widely known logic programming language.

Prolog places the following restrictions on the logic programming model:

- The order in which the goals are selected from the goal statement is fixed. Always the leftmost goal is selected first for satisfaction. This selection strategy is called *left to right depth first (LRDF).*
- If one unification fails, the textually next definition is selected for solution. This is called *shallow backtracking.*
- If no predicate matches (unifies) with the current goal, we undo the current derivation and we backtrack to the most recent choice point.
- Prolog has some extralogical features as well. These include arithmetic operators and I/O procedures, and their purpose is to make Prolog an efficient general purpose language.

The execution of a sequential Prolog program is clearly deterministic. The goals are satisfied left to right and the selection of the predicate to be unified with some goal is selected based on textual ordering.

The *search tree* of given Prolog program describes all possible ways of solving the initial goal statement. Each node of the search tree represents the set of "goals" that need to be satisfied for the initial goal to be true. Each branch of the search tree represents the selection of a goal for unification. A path from the root (the initial goal) to a leaf (Fail or Success) represents an possible execution of a Prolog program. The LRDF strategy corresponds to a path from the root to a "success" leaf in a depth first fashion, if there is such a leaf in the tree or a traverse of the whole tree if no such leaf is present. Generally, the structure of the

search tree is determined by the program structure, the initial goal statement, and the selection strategy.

Therefore neither And- nor Or- nondeterminism is present. Although this is bad news for parallelism, it is important because it gives the programmer a way to keep track of the program execution.

Operationally, each goal in Prolog is a procedure call. The clause

$$A_1 \leftarrow B_1, B_2, \cdots, B_n$$

is considered as a procedure definition. Unification realizes the common aspects of parameter passing, assignment, data selection and construction[Shapiro, 1989]. The Prolog programmer has in mind that goal B_1 will be solved before B_2 and goal B_2 will be solved before B_n. The speed of the program depends on the ordering of the predicates, so does program termination. For example, consider a logic program computing the factorial:

Example 7 `Consider the Prolog program that computes the factorial:`

```
Factorial(n,m) ← Factorial(k,l), Minus(l,m,1), Times(n,k,m).
Factorial(1,1).
Factorial(0,1).
```

This program will never terminate. It has the common problem of left recursion. One of the first tips a Prolog programmer learns is that you always put the exit condition first. It is explicit in his mind that the interpreter will first look at the first definition and only in case of failure will proceed to the next one.

Reintroducing nondeterminism to Prolog programs for the sake of parallelism is not as straightforward as it may seem at first. Prolog is an incomplete proof procedure for logic programs: the interpreter may fell into an infinite loop, when there is a way to give an answer. Prolog programmers know that and their programming practice is tightly coupled to the LRDF execution. Even if this practice changes for the sake of efficiency in traversing the tree the programmer should have a way to keep track of control flow in the program.

We refer to *Pure Prolog* as the version of Prolog that lacks any extralogical elements and explicit constructs for the control of backtracking, like the "cut" mechanism. Pure Prolog being simple and straightforward, is simple to reason and to exploit for parallelism implicitly.

Sequential Prolog interpreters have used the sequential execution assumption in an attempt to achieve faster execution. Stripping the language of all these bells and whistles makes the exploitation of parallelism almost straightforward.

3.3 Characteristics of Logic Progr. languages

The abstract logic programming model has a number of distinctive features the most striking of which is nondeterminism. Many abstract computational models besides the logic programming model are nondeterministic including Turing machines and nondeterministic finite automata. Reactive systems, the systems that maintain some interaction with their environment are also nondeterministic. The two forms of nondeterminism are different and they are frequently distinguished in the literature[Shapiro, 1989]. The first type is called "don't know" nondeterminism, while the second "don't care" nondeterminism. In case of "don't know" nondeterminism the programmer does not need to know which of the choices specified in the program is the correct one. He only cares whether there is a nonempty set of satisfying choices. The problem with "don't know" nondeterminism is that you cannot tell anything about the computation before it completes. On the other hand "don't care" nondeterminism allows the production of partial outputs, irrespectively of whether the computation will succeed or fail in the end.

The ability of the abstract programming model to express both of these interpretations of nondeterminism is of major importance. It allows the specification of *stream parallelism.* Stream parallelism allows to agents to share incomplete results, thereby incorporating the notion of pipeline.

Logic programming languages correspond have inherent the notion of a stack, in contrast to functional languages which have inherent the notion of the reduction graph. Significant research work has been done on of efficient memory management for logic programming languages [Tick, 1988; Dobry, 1990] with impressive results.

They also have the single assignment property. Each variable is bound at most once during the program execution, via a substitution. It may be the case that in program execution a variable remains unbounded if no goal satisfaction imposes a restriction on it.

However, the most important characteristic of logic programming languages is the use of unification as the primary operation. Unification plays the role of pattern matching in functional languages and has the "innocent" side-effect of specifying the satisfying substitution or assignment for the program.

3.4 Parallel execution of Logic Progr. Languages

Most logic languages are interpreted. Therefore a lot of research has been done on the problem of building parallel interpreters for logic languages. Here we will examine Pure Prolog parallel interpreters. The traditional Prolog interpreter[Clocksin and Melish, 1981] explores the solution space in a left to right, depth first way (LRDF). A parallel interpreter that generates exactly the same set of solutions as a sequential LRDF one is called *acceptable*, while an interpreter that generates the solutions in the same order as the sequential one is called *conservative*.

The granularity of parallelism achieved by a parallel Prolog interpreter depends on the type of the tree that represents the search, on the control strategy followed, and on the parts of the interpreter where parallelism has been incorporated.

There are four basic places of the interpreter where parallelism may be exploited: within the unification algorithm, among different unifications, data parallelism, parallelism in the control strategy. The grain of parallelism associated with the above stages is different and require separate handling as we will see in section 3.4.2.

3.4.1 The search tree

When we referred to the Prolog we described the search tree as a way to represent the search space of the Prolog program. The search tree provides only one way of describing the search space and it is not appropriate for describing every type and granularity of parallelism. In specific, it is able to describe Or-parallelism, but it is unable to describe And-parallelism.

To alleviate this problem the *And-Or tree* was introduced. The And-Or tree has two types of nodes:

- *AND-nodes* which represent an and-process which succeeds if all the children return "success".
- *OR-nodes* which represent an or-process which succeeds if at least one of its children processes returns "success".

The And-Or tree represents the static structure of the Prolog program with the initial goal statement. During program execution only the parts of the program that correspond to successful unification are generated.

The advantage of the And-Or tree is that it is able to describe different search strategies and support different granularities of parallelism. Its main disadvantage is that it needs much more memory to store.

From the basic And-Or tree, we can construct the LRDF And-Or tree which has additional arcs denoting the ordering between the AND-node children of an OR-node.

Other types of trees, variations of the ones above, have been developed as well, for the purpose of fitting the needs of special architectures or granularities of parallelism. Some of them are[Kacsuk, 1990]: the reduce-Or tree, the dataflow tree, the set-oriented tree, the or-forest tree, the and-process tree.

3.4.2 Levels of Parallelism

Parallelism may be exploited in different parts of the interpretation[Kacsuk, 1990]:

1. within the unification algorithm,
2. among different unifications,
3. data parallelism,
4. parallelism in the control strategy.

Parallelism within unification

When a clause heads has more than one literal, then the unification of the the corresponding arguments in the clause head and in the caller's goal can be performed in parallel. Parallelism within unification results in a conservative parallel interpreter because it does not affect the order we follow in searching the solution space.

This is an excellent although rather coarse source of parallelism as long as no shared variables are present. There are two approaches to handling the possible existence of shared variables:

- Perform argument unification in parallel and check the results for consistency.
- Check first whether any common variables are present, if the answer is yes perform unification of the terms sequentially.

The pursuit of parallelism at this part of the interpretation is worth while when deep and complex terms are unified, or when the average number of arguments in the clauses is relatively high. Studies on the behavior of Prolog programs [Jaffar, 1987] showed that the average number of clause head arguments in most Prolog programs is about three, which is relatively low. This justifies the choice of the second approach to the problem of shared variables, we cannot afford to synchronize. The synchronization cost would wipe out the gain from the parallel unification of the arguments.

Parallelism among Unifications

Multiple definitions may match a goal and a sequential Prolog interpreter will apply head unifications on them in textual order in case of failure of the first. With parallelism around we are able to do multiple head unifications in parallel. The interpreter continues to work when at least on of the unification is successful.

If the interpreter is conservative then priorities are attached to the interpreter processes and when more than one head unification is successful the one with the highest priority is selected for further resolution.

The performance of such an interpreter was analyzed [Kacsuk, 1985] and the results showed that it is advantageous to exploit parallelism at this level when the Prolog program contains large definitions. Database applications and generally fact intensive programs have this property. In this type of applications the majority of clauses are facts collected in large definitions.

Data Parallelism

It is very common in Prolog programs to associate common variables with the subgoals of a given goal as a means of explicit communication between them. The resulting configuration is very similar to a pipeline. The partial results produced by the first subgoal can be immediately consumed by the second subgoal.

This type of parallelism is easy to detect syntactically and is referred to as *stream parallelism* in the literature.

Parallelism in the control strategy

Up to now we have described only the LRDF control strategy and various parallelization tricks that one may apply to the basic algorithm without dropping this strategy at the cost of some slow down in execution time. The very next step is to take the full And-Or tree and solve the problem of breaking into pieces

so that we get the best performance at the lowest cost. In other words we need efficient algorithms for the parallel traversal of a search tree.

The corresponding strategies may be divided in two major classes:

- OR-strategies
- AND-strategies

OR-strategies

The subtrees starting from an Or-node describe the possible successful or failed resolutions of the statement represented by the node. If a separate interpreter process is associated is associated with each subtree then the interpreters are able to work in parallel. Notice that the source of Or-parallelism is the set of alternative clauses in the definition of a predicate.

The form of Or-parallelism may be *unlimited* if each possible subtree is handled by a different process or *restricted* if the number of simultaneously executing processes is limited.

The following restricted OR-strategies differ in the way they select the points where the interpreter is split into separate processes[Kacsuk, 1990]:

- *Eager AND-Process.* In an LRDF strategy a path of the And-Or tree is followed down satisfying the leftmost nonexamined And-child of an Or-node and all the Or-children of an And-node. When this process cannot proceed any further because of a "fail" the procedure backtracks to the last Or-node and tries the next to the right And-child. In the eager And-process strategy an And-node does not wait for a backtrack request from its parent Or-node but rather sends a backtrack request to its last Or-child and collects the result in a buffer for future reference. This parallel strategy may represent a conservative interpreter if the backup results are kept in LRDF order.

- *Lazy Or-Process.* In the lazy Or-process strategy an Or-node spawns a separate process for each one of its And-children. The first successful result is sent to its parent while the rest are kept in a buffer, to cater for backup results. When the father And-node requests a back-up result and the backup buffer is empty the Or-process reactivates the child processes.

- *Eager Or-Process.* The Eager or-process strategy is very similar to the lazy-or process. An Or-node spawns a separate process for each of its And-child nodes and stores the results received in a buffer. However in contrast

to the lazy-process the eager or-process will reactivate immediately the And-process that returned a successful result.

The applicability of the search strategies described above depends heavily on the Prolog program and the target architecture. It would be great if a tool, variant of a sophisticated compiler were developed which would analyze a Prolog program and given a target parallel architecture would decide which search strategy is most appropriate.

AND-strategies

The LRDF strategy always picks up the first one among the subgoals of the current goal for satisfaction. In a parallel architecture the subgoals may be satisfied in parallel. The only complication is that these goals may be related by common variables and while each goal is satisfiable by itself the relevant substitutions cannot be composed. The idea is that these subgoals share some form of environment. The decision on how to handle this environment defines the control strategy.

In the context of And-parallelism an And-process of the And-Or tree activates in parallel more than one child Or-process. If all the children Or-processes are activated then the And-parallelism is full, otherwise it is restricted.

In the case of restricted And-parallel strategies certain goals may be executed in parallel while others sequentially and an *ordering* mechanism determines the execution order of goals in each statement. The ordering mechanism classifies the restricted And-parallel strategies into *implicit* and *explicit.*

In implicit ordering of goals there is no language support for And-parallelism (Pure Prolog). The implicit ordering of goals is based on data dependency analysis of the goals in the clause bodies. Depending on when the major part of the data dependency analysis is performed we have compile time ordering of goals and run time ordering of goals.

In explicit ordering the programmer may use language annotations to support compile or run-time ordering of goals. If the programmer may use explicit goal ordering directives then we have *Control-flow ordering.* In other words the programmer is able to specify that a set of goals should be executed sequentially or in parallel depending on wether there are common variables. Parlog[Conlon, 1989] is a parallel programming language that supports explicitly AND-parallelism. In the clause bodies ";" and "," represent sequential and parallel execution mode respectively.

If the language supports declarations and annotations that explicitly declare input and output modes for clauses then we have *Explicit data-flow ordering of*

clauses. Each argument of a goal is marked as one of the following: "input", "output", or "don't know" depending on wether read and write operations are permitted on the argument. Parlog requires mode definitions for each clause in the program.

When the strategy pursues full And-parallelism, all goals of of a clause body are executed in parallel. The different goals may potentially produce different binding values for the same shared variable and a consistency check is needed. The consistency check may be done statically in which case when all solution for the goals are evaluated, the parent And-process takes into account only the consistent ones. It can be also done dynamically: the Or-processes send the results to the parent And-process which does the consistency check while the children Or-processes search fervently for more results. The basic problem with the exploitation of full And-parallelism is that the consistency analysis algorithm is very complex and time consuming. Besides that the And-processes should store at least temporarily a large number of partial results which will add significant overhead.

3.5 Why Logic Programming languages?

Logic programming languages are more high level than functional languages and they can be naturally implemented on a stack architecture.

They have a significant expressing ability and Prolog programs are relatively compact and easy to understand.

A number of specialized memory architectures have been developed specifically for logic programming languages which seem very promising.

Logic programming languages have a natural way to handle nondeterminism and they are able to describe easily reactive systems (systems that interact with their environment). Remember that this is a deficiency and a hot research issue in functional languages which are inherently deterministic.

3.6 Why not Logic Programming languages?

Logic programming languages are based on first order predicate calculus. As a consequence of that there are certain notions that cannot be handled elegantly in a logic programming language. "Negation" is such a notion. Prolog as a theorem prover has adopted the closed world assumption, namely whatever cannot be proven "true" is assumed "false".

The search of the problem space is another problem. Given a problem we do not have a specific algorithm to solve it, but rather we have to adopt a generic recipe of search for the satisfying solution. Unfortunately, there are no globally optimal approaches to searching. For every search strategy there may be a program that fails it.

3.7 Case Studies

Logic programming languages may be classified in two families in terms of parallelism:

- *Concurrent logic programming languages*
- *Parallel logic programming languages*

Concurrent logic programming languages explore "don't care" nondeterminism and contain augmentations to realize communication and synchronization. In these terms, a clause body that is atomic is viewed as a single process, while a conjunctive body is viewed as a process network, with the shared variables serving as communication channel between processes.

A program in these languages is a finite set of guarded clauses:

$$< head > \leftarrow < guard > | < body >$$

meaning $< head >$ is "true" if both the guards $< guard >$ and the $< body >$ are "true". Operationally to solve $< head >$ it is necessary to solve the guards in $< guard >$, and if their resolution is successful the subgoals of $< body >$ are solved in parallel.

The major type of parallelism implemented is "stream And-parallelism" according to which many subgoals can be executed in parallel. Examples of Concurrent logic programming languages are PARLOG[Conlon, 1989] and Concurrent Prolog[Shapiro, 1986].

Parallel logic programming languages are those that contain parallel data structures as well as parallel program structures. An example of such a language is DAP Prolog [P.Kacsuk, 1986; Kacsuk, 1990].

3.7.1 PARLOG

A PARLOG program is a finite set of guarded clauses of the form:

$$< head > \leftarrow < guard > | < body >$$

A PARLOG program defines operations that impose restrictions on the evaluation of clauses. For example $C_1 \& C_2$ means that the the clause C_2 is evaluated after the evaluation of clause C_1 has terminated. There are also two clause terminators that specify the form of the search: ";" the sequential search operator and "." the parallel search operator.

Modes are used in PARLOG to restrict access to variables. A mode declaration has to be included for each predicate and specifying the arguments as: input ("?") or output ("↑"). The modes are used to impose restriction on unification. Unification of a term with input mode means that its value can be substituted matching it with the values of the call. A call is suspended if a a variable with input mode has not been bound at input time. The clause is a consumer of the read variable. On the other hand an output variable has to be unbound within the call. A clause holding an output variable is considered a producer for that variable.

Example 8 Consider the following PARLOG program that searches a linear list for a certain element u:

(1) mode list-search(?, ?, ↑)
(2) list-search(u, [u|x], x).
(3) list-search(u, [v|x], y) ← u = / = v : search(u,x,y).

The mode definition for list-search appears on line 1. Notice that the first two arguments, the element and the list are declared input variables while the third argument is an output variable. Line 2 says that if u is the first element in the list, then output the part of the list on the right of u. Line 3 implements the recursive step by searching the tail of the list if the head of the list is different from u(notice the guard).

Chapter 4

Constraint Languages

4.1 The constraint paradigm

Constraint languages are by far the most high level of the language families we have examined. They are based on the notion of a constraint. A constraint expresses the desired relation between one or more objects. A constraint program defines a set of objects and relations between them. The ability to define an arbitrary set of constraints on a set of objects enables the programmer to express undecidable problems, or problems for which no solution is known.

Example 1 *One could disprove Fermat's last theorem by the following constraint program:*

```
exists x,y,z,n :  integer
       |
       (x^n + y^n = z^n  and
        n > 2)
```

This example points out the expressibility of the formalism. Note that Fermat's theorem has not been proved or disproved up to data, but is decidable in the sense that there is a Turing machine that can solve it (only we do not know which!).

In order to execute a program, the system has to solve the set of constraints defined by it, using a constraint satisfaction method. There is a variety of constraint satisfaction methods, with different execution, user participation and program layout methods. For example to solve a system of constraints the system may require that the user specifies an order in the constraints (Note that

the specification of an order in the constraints, is undesirable in case of parallel execution), or annotations that would help efficient code generation. We will present different constraint satisfaction methods on section 4.2

Logic and functional languages have the ability to express constraints but constraints are not the central notion in the language. Constraints have a conceptual elegance because they can express objects and their behavior briefly, and simply. They also have great potential for abstraction, which is a notion a bit stretched in the other families. As in the functional paradigm functions are first class objects, in the constraint paradigm constraints (bi-directional equalities) are first class objects (i.e we can define higher order constraints in terms of simpler ones).

The following displays a nice aspect of constraint languages :

> Constraint programs specify a set of objects and their behavior via dependencies on the environment and on other objects.

This is very important, when considering parallelism because no extraneous dependencies are forced, by the language semantics or control constructs. This does not mean that there cannot be bad constraint programs. The purpose is to give the programmers the means to express themselves in a more dependency oriented way.

Constraint languages are by virtue single assignment, the values of the variables are established through the satisfaction of constraints and therefore the idea of multiple variable assignments does not exist in the language. No explicit notion of storage is present. Emphasis is given on problem solving.

These advantages of constraint languages introduce many inefficiencies in their implementation, which we will discuss in section 4.5. In a few words the abstractions they give to the user are so powerful that the inefficiencies of implementation are more apparent than those of the other declarative language families.

4.2 Constraint satisfaction techniques

Constraint satisfaction techniques range from those assuming an order for the constraints and satisfying them one after another, to those whose solution involves solving algebraic systems of concurrent equations and finding the optimal selection of the problem parameters.

The problem of constraint satisfaction is a very difficult one and most powerful solutions are weak methods, namely tend to be domain dependent which

limits their applicability. These methods take advantage of the inherent characteristics of the problem at hand to solve the problem efficiently.

Here we will examine general methods or heuristics which although suboptimal result in "acceptable" speed.

We should note however, that there is no reason why a system should limit itself to a single constraint satisfaction method. In the ThingLab [Borning, 1981] an experienced programmer can built a "simulation" a small constraint satisfaction system for solving limited classes of problems defined by objects and constraints.

Generally a constraint program can be represent as a graph. Nodes represent constraints. When two constraints interact they are connected. The arcs of the graph denote the direction of information flow. Constraints are bidirectional in their general case which means that the constraint graph is undirected. Most constraint methods operate based on this graph and the trade off present is how to get the best results considering as small a part of the graph as possible.

Local Propagation

Local Propagation is one of the simplest constraint satisfaction mechanisms [Steele, 1980]. The mechanism is similar to the principle of data flow with minor differences, due to the bidirectionality of constraints. In the beginning values are broadcasted, through the connections. When a constraint has sufficient information it "fires" calculating one or more or the values for its variables. These values are in turn broadcasted and so on, until all constraints are satisfied, or found to contradict.

Usually constraints are tried in textual order (alternatively the user may define an order in which constraints should be satisfied.) when they are satisfied sequentially. We should note however that this is only an operational choice; constraints are by nature, bi-directional and do not have a prespecified application order, apart from that implied by the data dependences of the problem.

If we want to apply local propagation in parallel, the considerations that apply are similar to those in case of parallel graph reduction. We have to trade off between the time required to satisfy a constraint and the cost of setting up a process added to the communication cost.

The problem with local propagation is that it is *local* in nature. Each node has only local information, namely information provided by the arcs connected to that node. It has no knowledge of the graph structure. For example if a node is part of a cycle local propagation will fail to satisfy the constraints because it does not know that the constraints are related. This in effect, means that the

method is unable to solve systems of simultaneous equations.

An advantage of local propagation is that the system can keep track of how a value was acquired.

Example 9 Suppose we had to satisfy the simple constraints:

$$3 + p = 4 + s, 4 * s = 3$$

The second constraint fires immediately, as it has locally all the necessary information. Then it distributes the value of s to the first constraint and the first constraint can now be satisfied. The system can keep track on who gives values to whom, so that in case of time varying constraints it can simply repeat the firing[1] pattern, and save time by checking it instead of finding it again.

Relaxation

A way to solve constraint programs that involve simultaneous numerical equations is the classic method of iterative numerical approximation technique, **relaxation**. As a consequence of this it this method is applicable only in objects with continuous numeric values, not Boolean or integers.

Relaxation makes some initial approximation on the values of objects, possibly defaults. Then the error of this assumption is evaluated and new values are estimated, in an aim to minimize the error.

The underlying assumption of relaxation is that the error for the program objects can be reasonably approximated by some linear function. To determine the new value approximation we perform a *least square fit* on the the error functions of the objects. This carries the assumption that the constraints on the values of the objects can be adequately approximated by a linear equation.

The problem with relaxation is that this process does not converge in all cases.

The most important problem however is that it is *slow*.

Propagation of degrees of freedom

Relaxation is useful when local propagation fails, because of a cycle in the constraints. However the cost of relaxation is high. You have to iterate on a procedure of assigning values, propagating them, estimating the error, changing the values and repeat. If the graph is really large we do extra work on pieces of it, where local propagation would be fine.

[1] Here we use the notion of firing and satisfaction interchangeably for the notion of satisfying a constraint.

If we compute the *degrees of freedom* of every variable then, we can prune those parts of the graph that do not participate in a cycle and apply relaxation just on the cycles. Here is where *compilation* would help. Program names can be classified according to their degrees of freedom and names with enough degrees of freedom together with their associated constraints be removed from the graph. From the results of relaxation, we establish a deduction step that establishes the value of the rest of the names.

How do we compute the number of degrees of freedom of a name from a set of constraints? This problem is not trivial. A heuristic used by some systems is: "a name with a single constraint attached to it, has enough degrees of freedom".

Propagation of degrees of freedom is not as powerful as propagating known states because we do not use information attached to the arcs, in other words we do not use information related to the type of the relevant constraints.

Graph Transformation

Another technique for constraint satisfaction is *Graph transformation.* One of the problems with the techniques we have described so far is that they do not have enough nonlocal knowledge of the constraint graph. For example local propagation is limited to examining only one node and the arcs connected to it. By transforming the graph, a node in the transformed graph may represent a larger part of the initial graph and therefore more global knowledge.

The problem with graph transformation techniques is that they are still local in nature. They take into account a bigger part of the graph but they are unable to cope with system of simultaneous equations. The difficulty involved in cycles developed out of systems of simultaneous equations, is that generally they are non local in extend, and therefore they require sophisticated equation solving techniques.

Other Constraint Satisfaction Techniques

The methods we examined in this section, do not cover by all means the whole span of constraint satisfaction methods. There are a number of constraint satisfaction techniques that come from the field of Artificial Intelligence, efficient heuristics for searching and generally theorem proving systems.

The general methods we examined are related to constraints of general type. What if we have constraints attached to the constraint satisfaction mechanism (*metaconstraints*) or constraints expressed in terms of constraints(*higher order constraints*)?

A difficult problem is that of time varying constraints, constraints that involve time. In this case the constraint graph is dynamic. The changes in the constraint graph can be handled by a form of propagation called *retraction.* Retraction works by retracting a value that has changed in the constraint graph and appropriately changing the affected values. One can imagine how retraction might work in parallel starting from different sources in the constraint graph and slowly updating the graph. The efficiency of the procedure is constraint by the synchronization cost, but if the general structure of the constraint graph remains the same part of the work may be done at compile time. If not only values change in terms of time, but also constraints are inserted and removed as time passes, the problem of constraint satisfaction becomes more specific and should be solved by a method that "knows" a lot about the nature of the problem.

One should note however that even though we called the methods we described general purpose, they can not be used to solve arbitrary functions. This is a very limiting characteristic of constraint languages. A method to get around this problem that has been used is to built the constraint language on top of extensible languages like LISP or Smalltalk and integrate the extensibility mechanisms to the constraint satisfaction system.

A nice characteristic about local propagation and graph transformation is that they separate the control mechanism from the problem solving method. This is a very desirable characteristic because it allows to dynamically change one without affecting the other.

4.3 Case Studies in Constraint languages

Constraint satisfaction languages may be classified according to the mechanisms they use to break constraint cycles. There are those that use numeric techniques such as relaxation and those that use symbolic techniques to transform constraint programs containing cycles into equivalent programs that do not.

4.3.1 SKETCHPAD

SKETCHPAD[Sutherland, 1963] is one of the first constraint satisfaction systems for graphical applications. The key idea was to allow the user to draw a complex object by sketching a simple figure and them adding constraints to it. Objects previously drawn could be interned and turned into a macro. The system had a number of primitive constraints like making to lines parallel, perpendicular or of equal length. Constraints where represented internally as error expressions that evaluate to zero when the constraint is satisfied.

SKETCHPAD used propagation of degrees of freedom as its basic constraint satisfaction method and if this first method fails then it uses relaxation.

4.3.2 ThingLab

Thinglab[Borning, 1981] is a constraint simulation laboratory that builds the power of a SKETCHPAD-like system on top of Smalltalk[Goldberg and Robson, 1983]. An early version of the language described constraints in the same way as SKETCHPAD, but later the language was extended to allow constraints that included Smalltalk procedures.

ThingLab has also the ability to define higher order constraints and to use the constraint graphs interactively.

The constraint satisfaction methods used by ThingLab include "propagation of known states" in addition to the methods used by SKETCHPAD.

Recent extensions to ThingLab[Duisberg, 1986] allow it to deal with constraints that depend on time.

4.3.3 Steele's Constraint Language

Guy Steele's[Steele, 1980] constraint language was one of the first constraint systems based on algebraic manipulation of constraints. It used local propagation for constraint satisfaction and it also suggested the use of algebraic simplification techniques as well.

This system was storing information on how the constraints where satisfied, what rules were fired etc. and presented the user with explanation on its decision on various levels of detail. It also allowed the retraction of values so that changes could be incrementally computed without resolving the entire program.

4.3.4 Consul

Consul[Baldwin, 1989] is a general purpose constraint language based on Steele's Constraint language. It handles parallelism and experiments show that it can achieve significant speedup due to parallelism. More information on Consul can be find in the Appendix.

4.4 Advantages of Constraint Languages

Constraint languages are the most high level languages for the paradigms we considered in the declarative family.

They are very expressive, and concise which makes them easy to understand and reason about. Even logic programming languages which are more high level than functional languages are awkward in expressing some concepts. This is not the case with constraint languages they provide a natural way to express any problem.

4.5 Disadvantages of Constraint Languages

Although being high level a language helps the programmer thing in a more problem oriented way, it creates a lot of problems in implementation.

The problem of efficient execution is very tough, because it is frequently the case that a good constraint satisfaction technique for some problem is a very bad technique for some other problem.

As far as parallelism is concerned, things are not bright. The problem of constraint satisfaction is very difficult in itself and adding the problem of parallelism on to makes things even more complicated.

Chapter 5

Conclusions

We examined three different formalisms under the declarative paradigm focusing on issues related to parallelism.

The functional paradigm was the one closed to the imperative paradigm of the three. It attracted a lot of interest and significant results have been achieved. A striking advantage of it is that the programmer may debug a functional program on a single process, make sure it runs correctly and then run it on the multiprocess confident that it will run without significant complications. The major problems are the handling of the state of computation as well as determinism. Many attempts have been made to override the problem of determinism by innovative concepts such as continuations and environments. Functional programming turns out to be a very attractive solution in the case of closely coupled multiprocessors and small scale multiprocessors.

The logic programming paradigm based on first order predicate calculus is more high level, but it has an impressive treatment of "don't know" and "don't care" nondeterminism which make logic programming languages much suitable for system applications and stream parallelism. It is not clear whether logic programming is more appropriate for parallel applications than the functional model. It seems however, that both the models do not display large scale parallelism in terms of processes. The major source of fine grain parallelism is data parallelism. The more general form of parallelism exploited in both cases is some form of "argument parallelism" which is relatively each to check and apply.

Finally the constraint programming model, the most high level of all, seems the one with the most potential, but very limited work has been done in pure parallel constraint programming. A lot of work has been done in constraint logic programming and models which limit the power of the full constraints, in order

to get more efficient implementation.

The declarative paradigm presents an alternative solution to the problem of exploiting parallelism in programs adequately. Unfortunately the field is young and people are accustomed to a different model of computation, which is not only inadequate in terms of handling parallelism, but also is difficult to understand and naturally extend. Research in the area is relatively young and the programming community is very much attached to the imperative model. The declarative model slowly but steadily is developing.

Bibliography

[Abramsky and Hankin, 1987] S. Abramsky and C. Hankin, *Abstract Interpretation of Declarative Languages*, Ellis Horwood Limited, 1987,
(Chapter 2.1, 2.3: Formal methods for functional programs analysis. A good source of information on abstract interpretation, a technique used in functional language compilers for strictness analysis.).

[Allen and Kennedy, 1987] R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *TOPLAS*, Vol. 9(No. 4):pp. 491–542, October 1987,
(Chapter 1: Exploiting parallelism implicitly in parallel programs.).

[Appel and MacQueen, 1990] A. Appel and D. MacQueen, "Standard ML reference Manual," Technical Report ECS-LFCS-86-2 revised, Princeton University, 1990,
(Section 2.5: Manual of the latest version of ML.).

[Arvind, 1982] Arvind, "I-structures," Technical report, AI Memo, MIT, 1982,
(Section 2.1: A data structure for single assignment languages.).

[Arvind and Ekanadham, 1988] Arvind and K. Ekanadham, "Future scientific programming on parallel machines," In *Supercomputing: 1st International Conference Proceedings, Athens, Greece, June 1987.* Springer Verlag, June 1988,
(Section 2.1, 2.5.3: It contains a presentation of Id Nouveu a language for scientific programming.).

[Arvind *et al.*, 1989] Arvind, R. Nikhil, and K. Pingali, "I-structures: Data Structures for Parallel Computing," *ACM Transactions on Programming Languages and Systems*, Vol. 11(No. 4):598–632, October 1989,
(section 2.1, 2.5.4: They are structures that enforce single assignment. They were implemented on a tagged architecture in MIT.).

[Backus, 1974] J. Backus, "Programming Language semantics and closed applicative languages," *ACM Symposium on Principles of Programming Languages*, pages pp. 71–88, 1974,
(Section 2.1: Functional languages and program state.).

[Backus, 1978] J. Backus, "Can Programming be liberated from the von Neumann Style? A functional style and its Algebra of Programs," *Communications of the ACM*, vol. 21(No. 8):pp. 613–641, August 1978,
(Section 1.1,2.1: This is paper marked a start of a era for the functional paradigm. John Backus the person that designed Fortran and influenced the design of Algol, at his 1977 ACM Award lecture, stressed the need to adopt a different style for parallel programs. He proposed FP, a functional language to serve this purpose.).

[Baldwin, 1987] D. Baldwin, "Why we can't program multiprocessors the way we are trying to do it now..," Technical Report 224, University of Rochester, Department of Computer Science, August 1987,
(Chapter 5: It is a Comparison of the different formalisms in programming languages both within the imperative and the declarative paradigm against the challenge of parallelism. It points out the relative advantages and disadvantages and explains why the programmer should start thinking differently about programs. Differently means thinking in terms of dependencies as opposed to storage. It concludes that constraint programming strongest competitor to face this challenge).

[Baldwin, 1989] D. Baldwin, "CONSUL: A Parallel Constraint Language," *IEEE Software*, pages pp. 62–70, June 1989,
(Section 4.3: Constraint languages).

[Beer, 1989] J. Beer, *Concepts, Design, and Performance Analysis of a Parallel Prolog Machine*, Number 404 in Lecture Notes in Computer Science. Springer-Verlag, 1989,
(Section 3.4: Parallel execution of logic Programming languages).

[Bellia and Levi] M. Bellia and G. Levi, "The relation betweem logic and functional languages A survey.," *Journal of logic programming*, vol. 3:pp. 217–236,
(Chapter 5 It compares and contrasts the functional paradigm to the logic programming one.).

[Bird and Wadler, 1988] R. Bird and P. Wadler, *Introduction to functional programming*, Prentice Hall, 1988,
(Section 2.1 Nice introduction to functional languages.).

[Borning, 1981] A. Borning, "The programming aspects of ThingLab a Constraint Oriented Simulation Laboratory," *ACM Transactions on Programming Languages and Systems*, vol. 3, no. 4:pp. 353–387, October 1981,
(Section 4.3: Constraint languages).

[Borning *et al.*, 1990] A. Borning, B. Freeman Benson, and M. Wilson, "Constraint Hierarchies," Technical report, Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, Washington 98195, 1990,
(Variations to constraint programming).

[Borning *et al.*, 1989] A. Borning, M. Maher, A. Martindale, and Molly Wilson, "Constraint Hierarchies and Logic Programming," Technical report, Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, Washington 98195, 1989,
(Comparison of constraint programming with logic programming.).

[Carle *et al.*, 1987] A. Carle, K. Cooper, R. Hood, K. Kennedy, L. Torczon, and S. Warren, "A Practical Environment for Scientific Programming," *IEEEC*, 20(11), November 1987,
(Chapter 1: Exploiting parallelism implicitly in Fortran programs for scientific applications, the Rn programming environment.).

[Chen, 1987] M. Chen, "Can Parallel Machines be made easy to program? A data parallel model for functional languages," Technical Report TR YALEU/DCS/RR-556, Yale University, August 1987,
(Section 2.3: Programming the hypercube under the functional paradigm.).

[Church, 1941] A. Church, *The calculi of Lambda conversion*, Princeton University Press, Princeton NJ, 1941,
(section 2.1.1: Consistency of λ-Calculus.).

[Clark, 1979] K. Clark, "Predicate Logic as a computational formalism," Technical Report DOC 79/59, Department of Computing, Imperial College, London, 1979,
(Section 3.1: The abstract model of logic programming).

[Clocksin and Melish, 1981] W. Clocksin and C. Melish, *Programming in Prolog*, Springer-Verlag, 1981.

[Conlon, 1989] T. Conlon, *Programming in PARLOG*, Addison Wesley, 1989, *(Concurrent logic Programming languages).*

[Curry and Feys, 1958] H. Curry and R. Feys, *Combinatory Logic. Vol. 1*, North Holland, The Netherlands, 1958,
(section 2.1: curried functions).

[Darlinghton and Reeve, 1981] J. Darlinghton and M. Reeve, "ALICE: A multiprocessor reduction machine," In *Proceedings of the 1981 ACM Conference on Functional Programming languages and Computer Architecture, Portsmouth, NH.*, pages pp. 471–478, October 1981,
(section 2.3: Parallel graph reduction.).

[Dobry, 1990] T. Dobry, *A high performance architecture for Prolog*, Kluwer Academic Publishers, 1990,
(Section 3.4: Parallel execution of logic Programming languages).

[Duisberg, 1986] R. Duisberg, "Constraint-based animation: Temporal constraints in the Animus system," Technical report, University of Washington TR 86-09-01, 1986,
(section 4.1: Constraint languages).

[Field and Harrison, 1988] A. Field and P. Harrison, *Functional programming*, Addison-Wesley, 1988,
(Section 2.1 Introduction to functional programming using HOPE.).

[Flatt and Kennedy, 1989] H. Flatt and K. Kennedy, "Performance of Parallel Processors," *Parallel Computing*, 12(1):1–20, 1989,
(Chapter 1: Implicit parallelism and parallel programs, scalability synchronization speedup.).

[Gelernter, 1986] D. Gelernter, "Domesticating Parallelism," *Computer*, August 1986,
(Chapter 1: Gelender is among the implementors of Linda, a language based on message passing, following the imperative paradigm. He compares and contrasts the pursuit of parallelism explicitly and implicitly in programming languages, clearly favoring the first. He poses three interesting questions: Q: Does implicit parallel programming mean no more parallel algos? Q: How much

does the average programmer worry about parallelism? Q: Does declarative programming mean that we learn programming from the beginning?).

[Glaser *et al.*, 1984] H. Glaser, C. Hankin, and D. Till, *Principles of functional programming*, Prentice Hall International, 1984,
(Section 2.1: A simple book, on functional programming methodology. Good for start.).

[Goldbelg, 1988] B. Goldbelg, *Multiprocessor Execution of Functional Programs*, PhD thesis, Yale University, Department of Computer Science, April 1988,
(section 2.3: A compiler for the functional language AlfAlf was constructed for a hypercube. They achieved to reach the speed of compilers for imperative languages. The technique they adopted depends on Supercombinator analysis(expressing the program in terms of simple functions) and trading off between the cost of serial execution and the overhead involved in the creation of a process in a different node.).

[Goldberg and Robson, 1983] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Menlo Park, California, 1983.

[Gregory, 1987] S. Gregory, *Parallel Programming in PARLOG, the language and its implementation*, Addison Wesley, 1987,
(Section 3.7: Concurrent logic Programming languages).

[Gupta, 1986] A. Gupta, *Actor Systems*, The MIT Press, 1986,
(Chapter 1: a program methodology of loose control, within the imperative paradigm.).

[Harland, 1984] D. Harland, *Polymorphic Programming Languages, design and implementation*, Ellis Horwood Limited, 1984,
(Section 2.2: Deep exposition of polymorphism in programming languages.).

[Hill, 1974] R. Hill, "LUSH resolution and its completeness," Technical Report DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1974.

[Hillis, 1985] W. Hillis, *The Connections Machine*, MIT Press, 1985,
(Section 2.5: The connections machine is a SIMD parallel machine. It is constructed out of large numbers of processing elements and it is programmed in CmLisp which is an extension of Common Lisp with data structures and operations to domesticate concurrency.).

[Hoffman and Donnel, 1982] C. Hoffman and M. O' Donnel, "Programming with equations," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 1:pp. 83–112, January 1982,
(Chapter 5: A paradigm closely related to both the functional and the constraint one.).

[Hogger, 1982] C. Hogger, *Concurrent logic programming*, pages pp. 199–211, Academic Press, London, 1982.

[Hudac, 1986] P. Hudac, "Para-Functional Programming," *IEEE Computer*, August 1986,
(Section 2.4: Companion paper on parafunctional programming simpler than the first one. Presentation of ParAlf a parafunctional language.).

[Hudac, 1989] P. Hudac, "Concepts, Evolution and Application of Functional programming," *ACM Computing Surveys*, Vol. 21, No. 3, September 1989,
(section 2.1.1: λ-Calculus and functional languages.).

[Hudac and Goldberg, 1985] P. Hudac and B. Goldberg, "Distributed Execution of functional programs using Serial Combinators," *IEEE Transactions on Computers*, col. C-34, no. 10:pp. 881–891, October 1985,
(section 2.1 2.3: The notion of serial combinators is introduced as an extension of Hughes' Supercombinators. The basic idea is to derive the largest constructs in the user program that contain no concurrency.).

[Hudac and Wadler, 1988] P. Hudac and P. Wadler, "Report on the Functional Programming Language Haskell," Technical Report YALEY/DCS/RR656, Department of Computer Science, Yale University, 1988,
(Section 2.5: Haskell is considered the best in functional languages on today. It has all the nice characteristics that make it a general purpose language. Nevertheless, it is still a bit slow.).

[Hudak and Smith, 1986] P. Hudak and L. Smith, "Para-Functional Programming: A paradigm for Programming Multiprocessor Systems," In *Conference Record of the 13th ACM Symposium on Principles of Programming Languages*, January 1986,
(Section 2.4: This paper examines the possibility of augmenting a functional program with information for the compiler that will help it, improve his job of mapping the program on the target architecture. All this is done on the restriction that if the behavior of the program is the same with or without these augmentation).

[Hughes, 1989] J. Hughes, "Why functional programming matters?," *The Computer Journal*, Vol. 32, No. 2:pp. 98–107, April 1989,
(Section 2.6 : What are the advantages of functional programming.).

[Hughes, 1982] R. Hughes, "Supercombinators: A new implementation method for applicative languages," In *ACM Symposium of Lisp and Functional programming*, pages 1–10, 1982,
(section 2.3: How to extract program derivable out of a functional program.).

[ICOT, 1988] ICOT, editor, *Proceedings of the International conference on fifth generation computer systems.* ICOT, Tokyo, 1988.

[Jaffar, 1987] J. Jaffar, "On parallel unification for Prolog," *New generation computing*, Vol. 1(No. 5):pp. 259–279, 1987,
(Section 3.4: Exploiting parallelism in the execution of Prolog programs).

[Kacsuk, 1985] P. Kacsuk, "Parallel Unification Strategies for Prolog in Small size Multiprocessors," In *Proc. of the 4th Symp. on Microcomputer and Microprocessor Applications*, pages pp. 482–496, 1985,
(Section 3.4: Exploiting parallelism in the execution of Prolog programs).

[Kacsuk, 1990] P. Kacsuk, *Execution Models of Prolog for Parallel Computers*, The MIT Press, 1990,
(Section 3.4: Parallel execution of logic Programming languages).

[Kelly, 1990] P. Kelly, *Functional Programming for loosely coupled multiprocessors*, The MIT Press, 1990,
(section 2.3: Parallel graph reduction; A great presentation of the problems involved in compilation and parallel execution of functional programs.).

[Kowalski, 1974] R. Kowalski, "Predicate Logic as a programming language," In *IFIP 74 Amsterdam, The Netherlands: North-Holland*, pages pp. 556–574, 1974,
(The abstract model of logic programming was defined here.).

[Ladkin and Maddux, 1989] P. B. Ladkin and R. D. Maddux, "Parallel Path-Consistency Algorithms for Constraint Satisfaction," Technical report, International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley, California 94704-1105, August 1989,
(Section 4.2: Other Constraint satisfaction techniques.).

[Lamping, 1990] J. Lamping, "An Algorithm for Optimal Lambda Calculus Reduction," In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, January 1990,
(section 2.1: Lambda Calculus and functional programming languages.).

[Lloyd, 1987] J. Lloyd, *Foundations of logic programming*, Springer-Verlag, second edition, 1987,
(Section 3.1: The abstract model of logic programming).

[McCarthy, 1960] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I.," *Communications of the ACM*, Vol. 3, No. 4:pp. 184–195, 1960,
(Section 2.1, 2.5.4: Pure Lisp.).

[McGraw, 1982] J. McGraw, "The VAL language: Description and Analysis," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 1:pp. 44–82, January 1982,
(Chapter 2: One of the first Dataflow languages.).

[Milner, 1984] R. Milner, "A proposal for Standard ML," In *Proceedings 1984 ACM Conference on LISP and functional programming*, 1984,
(Section 2.5: Presentation of ML as a functional programming language.).

[Peyton-Jones, 1987] S. Peyton-Jones, *The implementation of Functional Programming Languages*, Prentice-Hall International, Englewood Cliffs, NJ, 1987,
(Section 2.3: Deals very deeply with the implementation issues in functional languages.).

[P.Kacsuk, 1986] P.Kacsuk, "The Design philosophy of DAP Prolog," Technical report, Queen Mary College, 1986,
(section 3.7: Case studies in logic programming languages).

[Polvika and Pakin, 1975] R. Polvika and S. Pakin, *APL: The language and its usage*, Prentice Hall, Englewood Cliffs, N.J., 1975,
(Section 2.5: APL was one the first algebraic language for arrays. It used essentially functional notation and although it had many imperative characteristics it greatly influenced the design of FP.).

[Ritchie and Thompson, 1974] D. Ritchie and K. Thompson, "The Unix Time sharing System," *Communications of the ACM*, Vol. 17(No. 7):pp. 365–375, July 1974.

[Roussel, 1975] P. Roussel, "Prolog: Manuel de Reference et d' Utilization.," Technical report, University d' Aix-Marseile, Groupe de IA, Marseile, France, 1975,
(Section 3.2: The definition of Prolog).

[Saraswat and Rinard, 1990] V. A. Saraswat and M. Rinard, "Concurrent Constraint Programming," In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, January 1990,
(Variations to constraint programming).

[Schnorf and Ganapathi, 1989] P. Schnorf and M. Ganapathi, "Compilation of Single Assignment Languages: Analysis and Propositions," Technical Report CSL-TR-89-399, Stanford University, November 1989,
(Section 2.1: The report deals with the inefficiency problems introduced in the compilation of single assignment languages. It does not deal with advanced characteristics of functional languages, like higher order functions and lazy (call by need) semantics. It aims at reducing the inefficiency that multiple copies introduce in single assignment languages.).

[Shapiro, 1986] E. Shapiro, "Concurrent Prolog: A Progress Report.," *IEEE Computer*, vol. 19:pp. 44–58, August 1986.

[Shapiro, 1989] W. Shapiro, "The Family of Concurrent Logic Programming Languages," *ACM Computing Surveys*, Vol. 21(No. 3):pp. 413–510, September 1989,
(Section 3.7: Concurrent logic Programming languages: a variation of the logic programming model).

[Silbey *et al.*, 1986] A. Silbey, V. Milutinovic, and V. Mendoza-Grado, "A survey of advanced microprocessors and HLL Computer Architectures.," *IEEE Computer*, pages pp. 72–85, August 1986,
(Chapter 1: Architectures and parallelism).

[Steele, 1984] G. Steele, *Common Lisp*, Digital, 1984.

[Steele, 1980] G. L. Steele, *The definition and implementation of a computer programming language based on constraints*, PhD thesis, MIT, AI Lab, August 1980,
(section 4.3: Steele's constraint language).

[Stoughton, 1988] A. Stoughton, *Fully Abstract Models of Programming Languages*, Wiley, 1988,
(Chapter 1: Different models for programming languages.).

[Sutherland, 1963] I. Sutherland, "SKETCHPAD: a Man-Machine Graphical Communication System," In *Proceedings of the Spring Joint Computer Conference.* IFIPS, 1963,
(Section 4.3: Constraint languages).

[Talia, 1990] D. Talia, "Survey and Comparison of PARLOG and Concurrent Prolog.," *SIGAPLAN Notices*, Vol. 25(No. 1), January 1990,
(Section 3.7: Comparing formalisms under in the family of concurrent logic programming languages).

[Tick, 1988] E. Tick, *Memory Performance of Prolog Architectures*, Kluwer Academic Publishers, 1988,
(Section 3.4: Parallel execution of logic Programming languages).

[Treleaven, 1990] P.C. Treleaven, editor, *Parallel Computers, Object-Oriented, Functional, Logic*, John Wiley and Sons, 1990,
(Chapter 1: General background text on the different programming language paradigms.).

[Turing, 1937] A. Turing, "On computable numbers with an applications to the entscheidungsproblem.," *Proc. London Mathematical Society*, Vol. 42:pp. 230–265, 1937,
(section 2.1.2: Presentation of the Turing Machines).

[Turner, 1989] D. Turner, *Research topics in Functional programming*, Addison Wesley, 1989,
(Section 2.2; 2.3: General information on the state of the art in functional programming.).

[Turner, 1981] D. Turner, "The semantic elegance of applicative languages.," In *Proceedings of the 1981 ACM Conference on Functional Programming and Computer Architecture, Portsmouth, NH.*, pages pp 85–92, October 1981,
(section 2.6: Why functional languages?).

[Turner, 1985] D. Turner, "Miranda: A non-strict functional language with polymorphic types," In *Functional Programming Languages and Computer Architecture*, pages pp. 1–16. Springer-Verlag, New York, September 1985,
(Section 2.5 : A different functional language.).

[Vegdahl, 1984] S. Vegdahl, "A survey of the proposed architectures for functional languages," *IEEE Transactions on Computers*, C-33:12:pp. 1050–1071, December 1984,
(Section 2.3: A overview of the contemporary architectures and their suitability of functional languages.).

[Wadge and Ashcroft, 1985] W. Wadge and E. Ashcroft, *Lucid, the Dataflow Programming Language*, Academic Press, 1985,
(section 2.5: The dataflow paradigm is closely related to the functional one although it shares some features with the constraint paradigm as well.).

[Wei and Gaudiot, 1988] Y. Wei and J. Gaudiot, "Demand interpretation of FP programs on a Data-Flow Multiprocessor," *IEEE Transactions of Computers*, vol. 37, No. 8, August 1988,
(Section 2.3, 2.6: They make a good case why the functional paradigm is worth pursuing in an imperative world).

[Williams, 1982] J. Williams, "On the development of the Algebra of functional programs," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 4:pp. 733–757, October 1982,
(Chapter 2: Modeling functional programs).

[Wilson and Borning, 1989] M. Wilson and A. Borning, "Extended Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison," Technical Report 89-05-04, Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, Washington 98195, July 1989,
(Variations to constraint programming).

Part II

A Front End for CONSUL

Chapter 6

Introduction

CONSUL [2] is an experimental constraint language designed for general purpose programming. The original motivation for the language was to study implicit parallelism, but the language turned out to be of more general interest. Many programs have been successfully written in a variety of application areas. As these applications were being written, it became clear that a more user friendly syntax for the language was needed.

Constraint programming is a descendant of logic programming in that any constraint corresponds to a logic predicate. However constraint languages have certain relations built in in the form of primitive constraints. The primitive constraints in CONSUL represent basic arithmetic, comparisons and set operations. Because these primitives are so simple, programs written in raw CONSUL[1] tend to be big. A lot of temporary variables and quantifiers are needed to express constraints at the level users find most natural.

The specifications set for the front end were:

- *Simplicity.* The front end should be simple enough for programmers to understand and become comfortable with it in a short time. This means that the notation should be familiar, either from set theory (if possible) or from widely used programming languages.

- *Compactness.* Expressions in the language should be compact, so that even powerful programs are short enough to be readable.

[1] By *raw CONSUL*, we mean the language accepted by the original CONSUL interpreter, as opposed to the language of the front end. From now on the term CONSUL will refer to the language of the front end.

- *Separability.* The front end should enable the programmer to split a program across multiple files.
- *Structure.* The language of the front end should make the structure of CONSUL programs clear.
- *Semantic Checking.* Programmer errors that can be detected by the front end should be, in order to speed up program testing.

The front end has three phases. The first phase, parsing, parses the program into a syntax tree, the second one, optimization, applies basic optimizations to it, while the last one, code generation, generates raw CONSUL code. The syntax tree representation looks pretty much like raw CONSUL: the features of CONSUL that do not have direct analogs in raw CONSUL have been removed and the infix notation has been turned into Polish. The optimization phase performs constant folding and constant expression evaluation. Code generation breaks the s-expressions of the syntax tree into raw CONSUL by introducing the necessary quantifiers and temporary variables.

A number of possible extensions to this work lie ahead. One possibility is optimizing the code produced by the front end. The idea is to extend the limited optimization performed at present to a full blown optimization. There is indication that analogs of most imperative language optimizations apply to CONSUL. However the declarative nature of CONSUL means that the techniques for applying these optimizations to imperative languages won't necessarily work for CONSUL. For example, terms like data flow analysis are not clearly defined for CONSUL.

Another extension to the front end would be adding annotations and/or hints on how to parallelize the program. This could be helpful to compilers for CONSUL.

Finally new features could be added to the language via macros, new statements, or libraries. The need for such features will evolve as the front end is used in extensive programming.

Chapter 7

Raw CONSUL

CONSUL is very much an experimental prototype of a constraint language. It supports demonstrations of realistic programs for a variety of applications, but in a laboratory setting rather than industrial production. This chapter describes raw CONSUL, the language accepted by the original CONSUL interpreter.

CONSUL's formal basis is in set theory. Formally, everything in CONSUL is a set. Thus, the fundamental data type is the set and the fundamental operators are the logical connectives (and, or, not) and quantifiers. However a number of abstractions have been provided to make the language easier to use than raw set theory. For example, there are a number of built in data types like sequences, integers, characters et cetera. Each of these data types has a formal set theoretic definition, but most programs need not deal directly with this aspect of CONSUL. Each built in data type is associated with built in relations that correspond to common operations on that type. For example, simple comparisons and arithmetic relations are defined for integers.

As an example of raw CONSUL, consider computing the GCD of two numbers. There are at least two ways to do this. The first is to give a direct, very declarative, translation into raw CONSUL of the definition of GCD, namely, the GCD of two integers b and c is the positive integer a that divides both b and c such that there is no greater integer with the same property. The second way is to compute the GCD using Euclid's algorithm. Figures 7.1 and 7.2 implement these two approaches. Figure 7.1 uses the declarative approach while figure 7.2 uses Euclid's algorithm.

The two raw CONSUL programs illustrate some of the ideas behind CONSUL. In figure 7.1 the user defined relation *dec_gcd* (lines 1–12) is the actual definition of the GCD. Line 1 defines *dec_gcd* as a subset of the cross product of

three integers. This definition reflects the fact that relations are formally sets. "Subset" and "cross" are examples of so called set constructors. Set constructors are raw CONSUL's way of allowing programmers to write constant sets. Line 2 introduces the formal parameters of the relation, *gcd*, *b* and *c*, while lines 3–12 specify the constraints that *gcd*, *b* and *c* must satisfy for *gcd* to be the GCD of *b* and *c*. Lines 4–5 state that *gcd* is a common divisor of *b* and *c*. Lines 6–12 state that there exists no common divisor of *b* and *c* that is greater than *gcd*. All these constraints ought to hold simultaneously, as indicated by the "and" on line 3. Lines 13–26 form the main part of the program that relates input to output. The existential quantifier in line 13 introduces the variables for which the problem has to be solved.

Input and output are modeled as sequences. Variables that represent these sequences are bound to external files by the "input" and "output" constraints (lines 21 and 26). Lines 23 and 24 constrain *b* and *c* to be the first and second element of the sequence *in* respectively. Finally, line 25 declares that the output is the GCD of the inputs.

Figure 7.2 is similar to Figure 7.1, except that it uses a different definition of GCD (*rec_gcd*). Note that relations in CONSUL can be defined recursively, as shown in line 11. The main part of the program (lines 12–31) is changed to reflect that *b* has to be greater than *c* for Euclid's algorithm to work. This is done by the "or" on lines 24–30.

The following features are the main reasons why CONSUL is well suited for general purpose programming:

- Set theory provides data structures (sets), control structures (logical connectives) and block structures (quantifiers), that are all independent of execution order. Within these basically unordered semantics, sequences provide a way to describe sequential ordering when it is needed. These are the main sources of parallelism in CONSUL.

- Different programming styles are supported by CONSUL, ranging from highly declarative (Figure 7.1) to highly algorithmic (Figure 7.2). This gives programmers the ability to control program efficiency through proper choice of algorithms without leaving the declarative framework.

- Being a constraint language, CONSUL can express everything that can be expressed by predicate calculus.

After programming in CONSUL for a while, we realized that we needed a more compact syntax than the one illustrated in Figures 7.1 and 7.2. While

```
(1)  (define dec_gcd (subset (cross integer integer integer)
(2)      (rho (gcd b c)
(3)         (and
(4)           (remainder 0 b gcd)
(5)           (remainder 0 c gcd)
(6)           (not
(7)             (exists ((d (subset integer
(8)                             (rho (i)
(9)                                  (greater i gcd)))))
(10)                (and
(11)                  (remainder 0 b d)
(12)                  (remainder 0 c d))))))))

(13) (exists ((out (sequence integer))
(14)          (in (sequence integer))
(15)          (a integer)
(16)          (b integer)
(17)          (c integer))
(18)     (and
(19)       (size in 2)
(20)       (size out 1)
(21)       (input in "stdin")
(22)       (elt a out 0)
(23)       (elt b in 0)
(24)       (elt c in 1)
(25)       (dec_gcd a b c)
(26)       (output out "stdout")))
```

Figure 7.1: Definition of *gcd* in raw CONSUL

```
(1) (define rec_gcd (subset (cross integer integer integer)
(2)     (rho (gcd b c)
(3)       (exists ((tmp integer))
(4)          (or
(5)            (and
(6)              (remainder 0 b c)
(7)              (equal c gcd))
(8)            (and
(9)              (remainder tmp b c)
(10)             (not-equal tmp 0)
(11)             (rec_gcd gcd c tmp)))))))

(12) (exists ((out (sequence integer))
(13)          (in (sequence integer))
(14)          (a integer)
(15)          (b integer)
(16)          (c integer))
(17)     (and
(18)       (size in 2)
(19)       (size out 1)
(20)       (input in "stdin")
(21)       (elt a out 0)
(22)       (elt b in 0)
(23)       (elt c in 1)
(24)       (or
(25)         (and
(26)           (greater b c)
(27)           (rec_gcd a b c))
(28)         (and
(29)           (less-equal b c)
(30)           (rec_gcd a c b)))
(31)       (output out "stdout")))
```

Figure 7.2: Euclid's algorithm for GCD in raw CONSUL

the semantics of raw CONSUL seem well suited to general purpose constraint programming, its syntax makes it hard to use. For example, defining relations as subsets of cross products is verbose and needlessly exposes programmers to the underlying formalism. Similarly, the use of "rho" in subset definitions is an artifact of the formalism that has little meaning to programmers. The built in constraints are so simple that constructing nontrivial relations requires many temporary variables. The quantifiers that introduce these temporaries break up the structure of programs, making them less readable.

The front end language evolved out of attempts to sketch CONSUL relations. When developing a relation, one wants to concentrate on its behavior, ignoring syntactic details. We thus found ourselves describing CONSUL programs using notations from set theory and familiar programming languages, instead of raw CONSUL syntax.

Raw CONSUL is now the intermediate code for CONSUL. This evolution reverses the usual situation, in which intermediate code is chosen after the source language. In the case of CONSUL, we first concentrated on getting a clean semantics and only afterwards worried about the representation. The clean semantics and simple syntax of raw CONSUL make it an ideal intermediate code, even if it is poorly suited to human use.

Chapter 8

The Front End

The front end extends raw CONSUL in two ways. First, it completely changes the syntax of the language. In particular, it provides infix notation, renames the primitive constraints, and changes the structure of relation definitions, quantifiers, and set constructors. These features provide compactness and improve the readability of programs.

Second, the front end adds features that make programs easier to maintain. A CONSUL program may be split into multiple files. Definitions may appear anywhere in a program. A definition that depends on other definitions may appear either before or after them, as long as the dependencies are not cyclic.

8.1 New syntax for existing features

The basic correspondence between the front end syntax and raw CONSUL is illustrated in Tables 1–7. A complete formal grammar for the front end language appears in the Appendix.

We wanted to use symbols for CONSUL constraints that were as close to standard set theory and programming languages as possible given the limitations of the computer keyboard. This explains the symbols chosen for basic arithmetic, comparisons, and subscription. Sets and subsets are enclosed in braces, reflecting standard mathematical notation. Similarly the size constraint is represented by vertical bars.

We borrowed our notation for conjunction and disjunction from Prolog (comma for "and", semi-colon for "or"). We borrowed C's exclamation point for negation. See Table 8.4.

The hardest constraints to find symbols for were those dealing with sets:

Constraint	*Raw Consul*	*Front End*
addition	(**plus** a b c)	a = b + c
subtraction	(**minus** a b c)	a = b − c
multiplication	(**times** a b c)	a = b * c
division	(**divide** a b c)	a = b / c
remainder	(**remainder** a b c)	a = b mod c

Table 8.1: `Correspondence of arithmetic operators`

Set Constraint	*Raw Consul*	*Front End*
subscription	(**elt** a b c)	a = b[c]
difference	(**set-minus** a b c)	a = b :−: c
union	(**union** a b c)	a =b **:U:** c
intersection	(**intersection** a b c)	a = b :*: c
size	(**size** b a)	a = | b |
index	(**index** pos pair)	pos = **index**(pair)
datum	(**datum** value pair)	value = **datum**(pair)

Table 8.2: `Correspondence of set constraints`

Comparison Constraint	*Raw Consul*	*Front End*
equality	(**equal** a b)	a = b
inequality	(**not-equal** a b)	a != b
greater	(**greater** a b)	a > b
less	(**less** a b)	a < b
less-or-equal	(**less-equal** b a)	a <= b
greater-or-equal	(**greater-equal** a b)	a >= b

Table 8.3: `Correspondence of comparison constraints`

Logic Operator	*Raw Consul*	*Front End*
and	(**and** $a_1 \cdots a_n$)	$a_1, \cdots, a_n$
or	(**or** $a_1 \cdots a_n$)	$a_1; \cdots; a_n$
not	(**not** a)	!a

Table 8.4: `Correspondence of logic operators`

Quantifier	*Raw Consul*	*Front End*
Existential	(**exists** $((v_1\ Set_1) \cdots (v_m\ Set_1)$ $\cdots$ $(v_k\ Set_n) \cdots (v_l\ Set_n))$ *body*)	**exists** $v_{n1}, \cdots, v_m$: $Set_1; \cdots; v_m, \cdots, v_l$: Set_n | *body*
Universal	(**forall** $((v_1\ Set_1) \cdots (v_m\ Set_1)$ $\cdots$ $(v_k\ Set_n) \cdots (v_l\ Set_n))$ *body*)	**forall** $v_1, \cdots, v_m$: $Set_1; \cdots; v_k, \cdots, v_l$: Set_n | *body*

Table 8.5: `Correspondence of quantifiers`

Set Constructor	*Raw Consul*	*Front End*
Set	(**set** $elt_1 \cdots elt_n$)	$\{elt_1, \cdots, elt_n\}$
Sequence	(**seq** $elt_1 \cdots elt_n$)	$<< elt_1, \cdots, elt_n >>$
Subset	(**subset** set restr)	$\{v_1 : S_1 ; \cdots ; v_n : S_n \mid$ restr$\}$
Powerset	(**powerset** set)	**powerset**(set)
Set of all sequences	(**sequence** set)	**sequence**(set)
Set union	(**set-union** $set_1 \cdots set_n$)	set_1 :**U**: $\cdots$:**U**: set_n
Set difference	(**set-difference** $set_1 \cdots set_n$)	set_1 :−: $\cdots$:−: set_n
Set intersection	(**set-intersection** $set_1 \cdots set_n$)	set_1 :∗: $\cdots$:∗: set_n
Cross product	(**cross** $set_1 \cdots set_n$)	set_1 :**X**: $\cdots$:**X**: set_n

Table 8.6: `Correspondence of set constructors`

I/O statement	*Raw Consul*	*Front End*
Input	(**input** seq "filename")	seq = **input** "filename"
Output	(**output** seq "filename")	seq = **output** "filename"

Table 8.7: `Correspondence of input/output constraints`

union, intersection, set difference, et cetera. In most programming languages these operations are overloaded on arithmetic symbols, for example, "+" for union, "−" for set difference, et cetera. Unfortunately, this approach doesn't work for CONSUL: Because every value in CONSUL is formally a set, programmers can apply set constraints to the same objects to which they apply arithmetic constraints. Set constraints and arithmetic constraints may have different solutions when applied to the same objects — for example, the integers 2 and 1 are formally the sets $\{\emptyset, \{\emptyset\}\}$ and $\{\emptyset\}$, respectively. The set difference of 2 and 1 is thus the set $\{\{\emptyset\}\}$, whereas the arithmetic difference is 1 (i.e., $\{\emptyset\}$). The symbols we finally adopted for set constraints are shown in Tables 8.2 and 8.6.

Note that some set constraint symbols are overloaded with set constructors. This overloading streamlines the language by providing a uniform notation for concepts for which raw CONSUL has multiple representations. Which raw CONSUL form is meant by an overloaded symbol can be determined by context.

In raw CONSUL, applying a set to one or more arguments constrains the tuple of arguments to be a member of the set. Since relations are sets, this convention is commonly used to "call" a relation. However, the same notation is used for other membership constraints. For example, if P is a set, then the raw CONSUL form "$(P\ \ s)$" asserts that s is one of the elements of P. Making membership assertions look like calls is one of the most baroque consequences of raw CONSUL's syntax. The front end provides the "IN" constraint as an alternative membership assertion. Thus, the above example could be written in CONSUL as "s IN P". Programmers usually think of calls and membership assertions as distinct things. The "IN" notation lets them clearly indicate which they mean. Furthermore, uses of "IN" are type-checked (the second argument must be a set), providing programmers an added measure of safety.

Operator precedence in CONSUL is similar to that in other languages. The default precedence for logical connectives is negation first, then conjunction, and finally disjunction. Precedence of arithmetic and set operators is summarized

Name	*Symbol*	*Precedence*
Cross product	:X:	1
Intersection	:*:	2
Union	:U:	3
Difference	:−:	4
Multiplication, Division	*, \	5
Remainder	**MOD**	5
Addition, subtraction	+, −	6

Table 8.8: `Precedence rules for arithmetic and set operators`

in Table 8.8. Low numbers indicate high precedence. The default precedence of any connective or operator can be overridden by parentheses.

CONSUL allows primitive constraints to be composed into elaborate expressions in a single statement. This composition eliminates the need for temporary variables that plagued raw CONSUL. The front end translates these statements into multiple raw CONSUL forms, adding the necessary temporary variables. For example, consider the geometric problem of finding the perpendicular bisector of the line segment between points (x_0, y_0) and (x_1, y_1). The solution is the set of points (x_r, y_r) that are equidistant from (x_0, y_0) and (x_1, y_1). From the formula for distance in a plane, x_r and y_r are solutions to the following CONSUL constraint:

$$(x_1-x_r)*(x_1-x_r)+(y_1-y_r)*(y_1-y_r) = (x_0-x_r)*(x_0-x_r)+(y_0-y_r)*(y_0-y_r)$$

The equivalent raw CONSUL code is much more verbose:

```
(exists
      ((a1 integer)
       (a2 integer)
       (a3 integer)
       (a4 integer)
       (a5 integer)
       (a6 integer)
       (a7 integer)
       (a8 integer)
       (a9 integer))
   (and
      (minus a1 x1 xr)
      (minus a2 y1 yr)
      (minus a3 x0 xr)
      (minus a4 y0 yr)
      (times a5 a1 a1)
      (times a6 a2 a2)
      (times a7 a3 a3)
      (times a8 a4 a4)
      (plus a9 a5 a6)
      (plus a9 a7 a8)))
```

Calls on user defined relations can be embedded in composite constraints. For example, the statement

$$d = a + f(\%, b, c)$$

corresponds to the raw CONSUL statements

```
(exists
      ((t1 integer))
   (and
     (f t1 b c)
     (plus d a t1)))
```

The idea is that that value "shared" between the called relation and its caller (t_1 in the example) is denoted by a "%", thus eliminating the need for a temporary variable. This notation was borrowed from Steele [6].

As another example of embedded calls, suppose we want to write a constraint that binds *sqmax* to the maximum of x_1, x_2, x_3, x_4, and x_5. We assume a relation "MAX(m,x,y)" which holds when m is the maximum of x and y. Since "maximum" is associative, we can nest calls on "MAX" to get the CONSUL constraint

MAX(sqmax, MAX(%, MAX(%, x_3, x_2), x_1), MAX(%, x_5, x_4)).

This example demonstrates calls both with and without the "%" notation. Writing the same relation without embedded calls takes more code and temporary variables:

exists a_1, a_2, a_3: integer
|
(MAX (a_1, x_3, x_2) ,
MAX (a_2, a_1, x_1) ,
MAX (a_3, x_5, x_4) ,
MAX (sqmax, a_2, a_3))

8.2 Program Structure

We have defined a structure for CONSUL programs that makes program organization clearer than it is in raw CONSUL. A CONSUL program consists of relation definitions, type definitions, constant definitions, and the main program body. Relation, type, and constant definitions must all come before the program body, but can appear in any order relative to each other. File inclusion statements allow parts of a program to come from different files.

Relations, types, and constants are defined by the "RELATION", "TYPE", and "DEF" statements, respectively. The syntax of "RELATION" is similar to that of procedure declarations in Pascal:

RELATION $name(v_{11},\cdots,v_{1n} : T_1 ; \cdots ; v_{l1}, \cdots, v_{lm} : T_l)$
body .

name is the name of the relation. The v_{ij} are the formal parameters, the T_i are their types. *Body* is the system of constraints that defines the relation. Type and constant definitions are written as

TYPE *name value.*
DEF *name value.*

name is the name that is defined while *value* is the type or constant it represents. These statements are discussed in more detail in Section 8.3.

The program body is introduced by the "MAIN" keyword:

MAIN
body .

File inclusion is controlled by the "INCLUDE" and "IFNDEF" statements, which are fully described in Section 8.3.

Figures 8.1 and 8.2 summarize the discussion so far by rewriting the GCD programs from Section 7 in the new CONSUL syntax. Figure 8.1 shows the declarative version; Figure 8.2 the algorithmic one. Note how much shorter these programs are than the originals, due to composition of constraints (e.g., Fig. 8.1 line 16; Fig. 8.2 lines 3, 11, and 12) and more concise forms for subsets (Fig. 8.1, line 5) and relation definitions (both programs, line 1). The new syntax should also be easier to understand, due to the clearer structure and notational similarity to conventional languages.

8.3 Extended Features

Up until this point, we have described front end features that do little more than provide better syntax for raw CONSUL functionality. There are also two areas in which the front end provides functionality that raw CONSUL does not have at all: File inclusion and static semantic checking.

File inclusion is done via the "INCLUDE" statement:

INCLUDE "filename" .

The effect of inclusion is the same as if the contents of "filename" had appeared in place of the "INCLUDE" statement. Note that the "MAIN" statement cannot be in an included file.

The "IFNDEF" statement makes managing multi-file programs easier. Its syntax is:

IFNDEF *name statement* .

```
(1)  RELATION dec_gcd(gcd, b, c:  integer)
(2)           gcd > 0,
(3)           b mod gcd = 0,
(4)           c mod gcd = 0,
(5)           !exists d :  {i :  integer | i > gcd}
(6)                   |
(7)                   (b mod d = 0,
(8)                    c mod d = 0).

(9)  Main
(10)     exists in, out :  sequence(integer)
(11)             |
(12)         (| out | = 1,
(13)          | in | = 2,
(14)          in = INPUT "stdin",
(15)          out = OUTPUT "stdout",
(16)          dec_gcd(out[0], in[0], in[1])).
```

Figure 8.1: CONSUL program that computes the GCD declaratively.

```
(1)  RELATION rec_gcd(gcd, b, c:  integer)
(2)             b mod c = 0, gcd = c
(3)           ; b mod c != 0, rec_gcd(gcd, c, b mod c).

(4)  Main
(5)      exists in, out :  sequence(integer)
(6)              |
(7)          (| out | = 1,
(8)           | in | = 2,
(9)           in = INPUT "stdin",
(10)          out = OUTPUT "stdout",
(11)          (in[0] > in[1], rec_gcd(out[0], in[0], in[1])
(12)           ; in[0] <= in[1], rec_gcd(out[0], in[1], in[0]))).
```

Figure 8.2: CONSUL program that computes the GCD using Euclid's algorithm.

CONSUL's "IFNDEF" is a simplified version of C's "#IFNDEF/#ENDIF": If *name* has not been defined when the "IFNDEF" is parsed, then *statement* is processed as if it had appeared in place of the "IFNDEF" form. If *name* is defined when the "IFNDEF" is parsed, then the entire "IFNDEF" statement is ignored.

Raw CONSUL has only one way to define a named constant. This mechanism exploits the fact that formally everything is a set to define constants, data types, and relations. However, it makes programs hard to read and vulnerable to inconsistent-usage bugs, because an object's definition cannot indicate how that object is to be used. The front end provides three definition forms, each indicating the intended uses of the defined object. Static checking ensures that defined objects are only used in the intended ways. The front end also detects cyclic definition errors (i.e., definitions that ultimately depend on themselves). Finally, uses of variables are checked for consistency with the variables' types.

Definitions may depend on other definitions. For example,

DEF *a* 1.
DEF *b* *a*.

is a legal series of definitions. Dependent definitions may appear in any order relative to each other; the front end will sort out the dependencies as it processes the program. As mentioned above, however, cyclic dependencies are errors.

The "TYPE" and "RELATION" statements assert that the defined name is a data type or relation, respectively. Attempts to use a name defined via "TYPE" as anything other than a type, or to do anything other than call a name defined via "RELATION", will be detected as errors.[1] "DEF" is used to define arbitrary sets, whose uses are not checked by the front end. "DEF" is normally used to define constants, although it actually provides the full power of raw CONSUL definitions.

Type checking in the front end is limited to checking that the actual parameters to user-defined relations have the same types as the corresponding formals. The front end also checks that user-defined relations are called with the correct number of arguments. It would be easy to extend these checks to include the arguments to primitive constraints as well.

We have limited type checking in the front end while we consider its interactions with the formal foundations of CONSUL. In some formal sense, there is

[1] As a minor exception to this rule, relations can also be passed as parameters to other relations.

```
(1) TYPE Ints POWERSET({1,3,5}).

(2) DEF Sum 8.

(3) RELATION Total ( Sum :  INTEGER; S : Ints)
(4)    S = EMPTY , Sum = 0;
(5)    EXISTS I : S
(6)       |
(7)       total(Sum-I, S :-:{I}).

(8) MAIN
(9)    EXISTS out :  SEQUENCE(Ints)
(10)      |
(11)      (total(Sum, out[0]),
(12)      |out| = 1,
(13)       out = OUTPUT "stdout").
```

Figure 8.3: CONSUL program that solves the 0-1 Knapsack problem

only one type for all CONSUL objects, namely "set". Many type checks are thus hard to justify formally. For example, quantifying a variable over the integer 7 is technically legal, albeit probably a programmer error. Correct handling of subtypes is another issue with which we are not yet comfortable. We believe that the present type checks could be strengthened without restricting the practical use of CONSUL, but we need to be careful in doing so.

As an example of a complete CONSUL program, Figure 8.3 shows a program that solves an instance of the 0–1 Knapsack problem. Specifically, given a set of integers ("$\{1,3,5\}$" on line 1), the program finds a subset of it whose members sum to "Sum" (line 2). Note how "TYPE" is used to define "Ints", which is subsequently used as a type in variable declarations, while "DEF" is used to define the constant "Sum". The subset found is placed on the standard output (line 13). The "Total" relation (lines 3 through 7) defines what it means for a set to sum to a value, while the main program (lines 8 through 13) calls "Total" and outputs the result.

Chapter 9

Implementation Issues

The front end has been integrated into a CONSUL interpreter running on Explorer™ Lisp Machines. Each of the three phases of the front end is discussed below. The discussion focuses on features unique to the front end. A great deal of standard compiler technology is also used, descriptions of which can be found in texts such as [1] [4].

9.1 Parsing

The first phase parses CONSUL programs into s-expression representations of syntax trees, using a recursive descent parser. It also expands "INCLUDE" and "IFNDEF" statements and checks for cyclic dependences between definition statements. Cyclic dependences are detected by the following algorithm:

1. For each name we have a data structure that contains the following items:
    - *Dependency Number.* The number of other definitions on which this one depends.
    - *Dependents List.* A list of names whose definitions depend on this one.
2. While parsing a definition, the parser forms a list of pending or still-undefined names on which that definition depends (the *Wait-For List*).
3. If the wait-for list is empty then
    - The newly defined name's dependency number is set to 0,

- The dependency number of each name in this one's dependents list is decremented, and
- For each dependent whose dependency number is now 0, recursively do step 3.

4. If the wait-for list is not empty, then
   - The new name's dependency number is set to the length of the wait-for list, and
   - The new name is added to the dependents list of each name in the wait-for list.
5. Cyclic dependences are evident at the end of parsing: Any name that still has a non-zero dependency number is involved in a cyclic dependence and is reported to the user.

As an example of definition tracking, consider the statements:

DEF a $b + c$.
DEF c $a - b$.
DEF b 4.

The first definition processed is a's. Since a depends on b and c, which have not been defined yet, its dependency number is set to 2 and it is put on the dependents lists of b and c. The definition of c similarly depends on b and a (which are still pending). Finally b is processed. It does not depend on any other definitions, but has a and c on its dependents list. Thus the dependency numbers of a and c are decremented. However, neither dependency number goes to zero, and so a and c cannot be processed further at this stage. When parsing is finished, a and c still have non-zero dependency numbers, and so must belong to a cycle.

9.2 Optimization

The optimization phase performs simple optimizations of constants. Additional optimizations may be added in the future, as discussed in Section 10.1. All optimizations are performed on the syntax tree produced by the parser, prior to raw CONSUL code generation.

The first thing the optimizer does is replace uses of names defined via "DEF" and "TYPE" by their definitions. Note that a definition may refer to other names (secondary names), which are in turn replaced by their own definitions. Secondary names are evaluated in the scope that was active when the primary name was defined. "DEF" and "TYPE" could also be translated directly into raw CONSUL "define" forms, avoiding the need for explicit substitution. However, explicit substitution makes program execution faster, because it eliminates the need to look up defined names in the run time symbol table.

After expanding any uses of defined names in an expression, the optimizer performs constant folding on the result. For now, only arithmetic expressions are subject to constant folding, but in the future we may choose to fold constant sets, characters, et cetera as well.

9.3 Raw CONSUL Code Generation

The s-expressions produced by the parser are almost usable as raw CONSUL code. The only difference is that the parser can produce arbitrarily deep sub-trees from composite constraints. These composites have to be broken into raw CONSUL's simple primitives, accompanied by a quantifier to introduce the necessary temporary variables. The code generator is simply a procedure that collapses composite sub-trees into raw CONSUL.

Syntax trees are collapsed into raw CONSUL by a post-order traversal. The traversal takes a syntax tree and returns two values. The first (*Forms*) is a list of CONSUL forms that need to be included in the final raw CONSUL,[1] the second (*Temps*) is a list of temporaries that have to be introduced to execute the raw CONSUL. The algorithm to traverse *Tree* is as follows:

1. If *Tree* represents an "exists", then its body is recursively traversed. An existential quantifier is created to be the returned *Forms*. The variables introduced by this quantifier are the union of those from *Tree* and the *Temps* list from the recursive traversal. The body of the new quantifier is the conjunction of the recursively generated *Forms*. The returned *Temps* is empty.

2. If *Tree* represents a "forall", then its body is recursively traversed. A new body (*NewBody*) is created as follows: If the recursively generated *Temps*

[1] Usually *Forms* contains a single raw CONSUL form that completely implements the tree; there are a few cases where *Forms* contains multiple forms that will serve as part of the raw CONSUL for a node at a higher level.

is nonempty then *NewBody* is an existential quantifier whose quantified variables are *Temps* and whose body is the conjunction of *Forms*. If *Temps* is empty then *NewBody* is just the conjunction of *Forms*. The body of *Tree* is replaced by *NewBody* and the result is returned, accompanied by an empty list of temporaries.

3. If *Tree* represents a connective, then the connected statements are recursively traversed. They are then replaced in *Tree* by the union of the generated *Forms* sets. The resulting tree is returned, accompanied by the union of the recursively generated *Temps*.

4. If *Tree* represents an equality statement, then do the following:

   - If both arguments to *Tree* are variables or constants, then *Tree* can be returned as *Forms*, with an empty list of temporaries.
   - If exactly one argument to *Tree* is a variable or constant, then recursively traverse the other argument. Replace the marker (see case 6 below) in the resulting *Forms* with the variable or constant argument. Return the result of this replacement, accompanied by the list of temporaries from the recursive traversal.
   - If neither argument to *Tree* is a variable or constant, then recursively traverse both arguments. Generate a new temporary. Replace the markers in the recursively generated *Forms* sets with this temporary. Return the union of the resulting *Forms* sets, building the accompanying *Temps* by adding the new temporary to the union of the recursively generated sets of temporaries.

5. If *Tree* represents a non-equality statement, then replace its arguments as follows:

   - Variables and constants need no replacement.
   - For each other argument, a new temporary is generated and the argument is recursively traversed. The argument in *Tree* and the marker in the recursively generated *Forms* are both replaced by the new temporary.

   Return the union of the modified *Tree* and any recursively generated *Forms* sets. The accompanying temporaries are those created above and any generated during recursive traversals.

6. If *Tree* represents an expression, then its raw CONSUL form is a primitive with one more argument than *Tree* has children. This extra argument represents the "value" of the expression. Process *Tree* as in step 5, adding a special marker to represent the "value" argument to the raw CONSUL primitive.

This algorithm is designed not to introduce unnecessary quantifiers or temporaries. The *Temps* result is a way of accumulating temporaries until either an existing existential quantifier is found in which to put them or creation of a new one is forced by a "forall". In this way our algorithm introduces temporaries via existing quantifiers, instead of creating new ones, whenever possible. The treatment of equality constraints avoids introducing unnecessary temporaries. It also avoids redundant raw CONSUL equality constraints. It does this by using a single temporary to represent the values of the expressions on both sides of the equality. Since both values are represented by a single name, equality is implicit in the semantics of CONSUL. As an example of code generation, consider the statement:

$$a * b = arrayA[j]$$

This is parsed into:

```
(equal
      (times a b)
      (elt arrayA j))
```

When the code generator encounters the "equal" node, it finds that neither child is a variable or constant. It thus recursively processes both children, producing raw CONSUL forms

(times *marker* a b)

and

(elt *marker* arrayA j)

A temporary is then generated to replace the markers, yielding

(times t_1 a b)

and

(elt t_1 arrayA j)

Code generation for the "equal" returns these two forms as its *Forms* result, and the list (t_1) as *Temps*. Code generation at higher levels of the tree will embed the two forms in an "and", and will appropriately quantify t_1.

The code generator infers the types of temporaries from the context in which they will be used: Temporaries used in arithmetic constraints must be integers. The types of temporaries used in set-related constraints are derived from the types of the other arguments to the constraint. Doing this may require deducing the type of a variable or constant. The type of a variable can be looked up in the symbol table. The types of integer and character constants are "INTEGER" and "CHARACTER" respectively; the types of constant sets or sequences are derived from the union of their element types. For example, the type of the constant

{1, 'c', *foo*}

is informally "set of type_of(1), type_of('c'), type_of(foo)" where "type_of(x)" denotes application of the typing algorithm to "x". In CONSUL this description is

powerset (type_of(1) :U: type_of('c') :U: type_of(foo))

Carrying out the "type_of" applications, and assuming that "foo_type" is the type of "foo", gives

powerset (INTEGER :U: CHARACTER :U: foo_type)

Finally, the type of a temporary introduced for a "%" sign in a relation call is just the type of the corresponding formal parameter.

Chapter 10

Extensions to the Front End

There are several areas in which the front end could be extended. Some of the extensions we are considering, and initial thoughts about their implementation, are discussed below.

10.1 Optimization

The raw CONSUL produced by the front end can be improved by optimizations analogous to those used in compilers for imperative languages. Constant folding is one example that is already done by the front end. Examples that would be useful but aren't yet implemented include eliminating multiple occurrences of constraints (analogous to common subexpression elimination), removing from "forall" bodies any constraints that do not depend on the quantified variable (loop invariants), et cetera.

Standard optimizations appear to be useful for CONSUL, but it is not clear that standard ways of performing them can be used. The standard conditions for validity of an optimization are based on data- or control-dependences between statements (or the absence of such dependences). Standard optimization algorithms are driven by analyses of these dependences. The notion of dependence, in turn, is based on the idea that statements will be executed in some order. However, execution order is irrelevant to constraint programs, so data- and control-dependence are ill-defined for them. For example, consider the CONSUL statements

$$x = 1, y = x + z$$

and

$$y = x + z, x = 1$$

Conventional copy propagation could turn the first into

$$x = 1, y = 1 + z$$

but could do nothing with the second. For CONSUL, however, the analog of copy propagation should be applicable to both sets of statements, since their textual order does not change the fact that x must be equal to 1 when it is added to z.

Another difference between optimization of constraint programs and of imperative ones is that a constraint may play different roles in a program at different times, or even multiple roles at once. For example, the constraint $x = 1$ might be a definition of x, or it might be a test of its value. Which role this constraint plays could depend on the heuristic used to execute the program in which it appears, on what variables were given as "inputs" to a particular run of the program, et cetera. To continue the earlier example, after copy propagation turns

$$x = 1, y = x + z$$

into

$$x = 1, y = 1 + z$$

a conventional optimizer might eliminate $x = 1$ as dead code. Definitions and some tests can be safely eliminated from a constraint program this way, but some tests cannot be. For example, eliminating the apparently dead equalities to x in

$$(x = 1, y = 1 + z); (x = 2, y = 2 * w)$$

produces a system of constraints with 2 solutions for y. If other constraints determine the value of x, one of these solutions will be valid for the original system and one won't be.

To purists, the whole idea of optimizing constraint programs is meaningless, since constraint programs are just mathematical statements that are either true or false. Nothing in the program itself determines how much time or memory is needed to prove truth or falsehood. Although this attitude is too extreme for real-world constraint programming, it does demonstrate an important point: execution efficiency depends on the *combination* of a program and its execution heuristic, and both must be included in any understanding of constraint program optimization.

The front end can produce better raw CONSUL than it does now, but several questions have to be answered first:

- What optimizations are legal for CONSUL's semantics, and under what circumstances?
- What optimizations make sense for the interpreter's execution method (or the execution methods of future compilers)?
- How does one analyze a CONSUL program to detect applicable optimizations?

The interpreter uses local propagation [5] to solve systems of constraints. This fact provides a starting point for answering some of these questions. Specifically, mode analysis algorithms for logic programming languages [3] might be adapted to provide data flow information about local propagation executing CONSUL. Well defined data flow eliminates many of the barriers to conventional optimization. For example, data flow implies information about the definers and users of values, so that the roles of constraints become clear; data flow also implies an order for computations. Much work remains, however, in determining what optimizations are compatible with CONSUL's semantics and whether mode analysis can provide accurate enough information to support these optimizations.

10.2 Annotations

Executing or compiling a CONSUL program requires solving certain problems whose solutions cannot be fully automated. Among these problems are solving the constraints in the first place, figuring out how to parallelize the program, et cetera. It currently seems that automatable heuristics will be able to solve these problems in many cases, perhaps even all cases that are practically relevant. However, it is possible that some sort of programmer assistance will eventually be needed. Annotations in CONSUL programs seem like a good way of providing this assistance.

We envision annotations as pragmas that can be attached to statements or groups of statements in a CONSUL program. These pragmas are hints at how a program should be compiled or executed, but should not affect the program's solutions. For example, annotations might indicate that a particular satisfaction heuristic should be used to execute a program, that certain parts of a program are good candidates for being solved in parallel with each other, et cetera. The meaning of a constraint program is more thoroughly separated from its execution than is the case in more imperative languages. Thus we expect that it will be relatively easy to develop annotations for CONSUL that let programmers control compilation and execution without altering the declarative meaning of programs.

The current CONSUL interpreter executes almost all programs without any guidance from the programmer. We hope that the same will be true of future implementations. However, we realize that this may not be possible as we try to transform CONSUL programs in more and more sophisticated ways. Annotations seem like a way of providing any guidance that turns out to be necessary without violating the language's fundamental semantics.

10.3 Higher Level Constraints

Designers of any language have to decide where the language's primitives end and where programmers have to start using those primitives to build their own abstractions. As we use the language, we may decide that flexibility or efficiency requires that the boundary change in CONSUL. There are several ways in which this could be done:

- New primitives could be added to raw CONSUL, with corresponding syntax added to CONSUL. This approach is appropriate where customized satisfiers for the new primitives are necessary.
- New features could be provided by macros in the front end. In other words, new syntax could be added to CONSUL, but the front end could handle this syntax by replacing it in-line with one or more existing raw CONSUL forms. This approach minimizes impact on CONSUL back ends, but macros will generally be solved less efficiently than built in primitives.
- User-defined relations that appear in many programs could be placed in libraries. At first libraries would be incorporated into programs via the "INCLUDE" statement. If libraries become widely used, other ways of linking them into programs may become necessary. For example, modules, ways of including only library relations that are actually called, and separate compilation into raw CONSUL would all be desirable in a sophisticated library system.

Chapter 11

Conclusions

We designed the front end to correct a number of deficiencies in raw CONSUL. The front end addresses these deficiencies as follows:

- The new notation is closer to that of mathematics and other programming languages, so programmers should find it simpler to learn and use than raw CONSUL.
- Infix notation, composition, and embedded relation calls make programs more compact than their raw CONSUL equivalents. Raw CONSUL forms that are extraneous to a program's real meaning are compacted out, thus improving program readability.
- "INCLUDE" and "IFNDEF" provide a way of dividing programs into multiple files.
- Distinguishing type, constant, and relation definitions makes program structure much clearer. Structure is also clarified by distinguishing the main body from the rest of the program.
- Semantic checking allows certain programming errors to be detected in the front end instead of during program execution.

Developing the front end also identified a number of exciting topics for further work. Among these are the use of raw CONSUL as an intermediate code, the possibility of optimizing it, and the role of semantic checking in CONSUL. Many of these issues are relevant to constraint languages in general, not only to CONSUL.

CONSUL is now a much more usable language than it was. This should make the language accessible to a wider user community and enable development of larger CONSUL programs. This, in turn, will facilitate experimental tests of CONSUL as a practical, general-purpose, constraint language.

Bibliography

[1] Aho, A. V., Sethi, R., and Ulman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Baldwin, D. "CONSUL: A Parallel Constraint Language". *IEEE Software*, June 1989 (**6:3**). pp. 62–70.

[3] Debray, S. "Static Inference of Modes and Data Dependencies in Logic Programs". *ACM Transactions on Programming Languages and Systems*, July 1989 (**11:3**). pp. 418–450.

[4] Fischer, C., and Leblanc, R. *Crafting a Compiler.* Menlo Park, Calif. Benjamin/Cummings, 1988.

[5] Mulac, J. and Baldwin, D. "Local Propagation as a Constraint Satisfaction Technique". Technical Report number 265, University of Rochester Dept. of Computer Science. Jan. 1989.

[6] Steele, G. "The Definition and Implementation of a Computer Programming Language Based on Constraints". Ph. D. Dissertation (Technical Report number AI–TR–595), Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Aug. 1980.

Appendix A

The grammar of the front end

The terminals of the grammar of the front end are ⟨id⟩, ⟨character⟩, ⟨integer⟩ and ⟨string⟩. An ⟨id⟩ is a sequence of characters that begins with a letter(a–z, A–Z), may contain digits 0-9 as well as the ASCII characters '_' and '#'. An ⟨integer⟩ is a signed or unsigned sequence of digits that does not start with 0. A ⟨string⟩ is any sequence of ASCII characters between double quotes. Finally a character is a single printable ASCII character surrounded by single quotes. Anything surrounded by /* */ is considered a comment and is ignored.

⟨program⟩ ⟼ ⟨driver⟩ |
⟨statement⟩ ⟨program⟩

⟨statement⟩ ⟼ ⟨relation⟩ |
⟨inclusion⟩ |
⟨type⟩ |
⟨define⟩ |
⟨ifndef⟩

⟨relation⟩ ⟼ RELATION ⟨id⟩ (⟨arglist⟩) ⟨body⟩ .

⟨inclusion⟩ ⟼ INCLUDE ⟨string⟩ .

⟨type⟩ ⟼ TYPE ⟨id⟩ ⟨vterm⟩ .

⟨define⟩ ⟼ DEF ⟨id⟩ ⟨exp⟩ .

```
⟨ifndef⟩ ⟼ IFNDEF ⟨id⟩ ⟨statement⟩ .

⟨driver⟩ ⟼ MAIN ⟨body⟩ .

⟨arglist⟩ ⟼ ⟨idseq⟩ : ⟨exp⟩ |
            ⟨idseq⟩ : ⟨exp⟩ ; ⟨arglist⟩

⟨idseq⟩ ⟼ ⟨id⟩ |
          ⟨id⟩ , ⟨idseq⟩

⟨body⟩ ⟼ ⟨and_term⟩ |
         ⟨and_term⟩ ; ⟨body⟩

⟨and_term⟩ ⟼ ⟨not_term⟩ |
             ⟨not_term⟩ , ⟨and_term⟩

⟨not_term⟩ ⟼ ! ⟨sterm⟩ |
             ⟨sterm⟩

⟨sterm⟩ ⟼ ⟨quantifier⟩|
          ⟨rel_stat⟩ |
          ⟨i/o-stat⟩ |
          ( ⟨body⟩ ) |
          ⟨set-memb⟩ |
          ⟨expression⟩

⟨quantifier⟩ ⟼ ⟨qname⟩ ⟨arglist⟩ '|' ⟨sterm⟩

⟨qname⟩ ⟼ EXISTS |
          FORALL

⟨set-memb⟩ ⟼ ⟨exp⟩ IN ⟨vterm⟩

⟨rel_stat⟩ ⟼ ⟨id⟩ ( ⟨paramlist⟩ )

⟨paramlist⟩ ⟼ ⟨exp⟩ , ⟨paramlist⟩|
              ⟨exp⟩

⟨expression⟩ ⟼ ⟨exp⟩ = ⟨exp⟩ |
               ⟨exp⟩ <= ⟨exp⟩ |
               ⟨exp⟩ >= ⟨exp⟩ |
               ⟨exp⟩ < ⟨exp⟩ |
               ⟨exp⟩ > ⟨exp⟩ |
               ⟨exp⟩ != ⟨exp⟩

⟨exp⟩ ⟼ ⟨pterm⟩ + ⟨exp⟩ |
        ⟨pterm⟩ - ⟨exp⟩ |
        ⟨pterm⟩

⟨pterm⟩ ⟼ ⟨vterm⟩ * ⟨pterm⟩ |
          ⟨vterm⟩ / ⟨pterm⟩ |
          ⟨vterm⟩ MOD ⟨pterm⟩ |
          ⟨vterm⟩
```

```
⟨vterm⟩ ⟼ ⟨uterm⟩ :−:  ⟨vterm⟩ |
          ⟨uterm⟩

⟨uterm⟩ ⟼ ⟨iterm⟩ :U: ⟨uterm⟩ |
          ⟨iterm⟩

⟨iterm⟩ ⟼ ⟨cterm⟩ :∗:  ⟨iterm⟩ |
          (  ⟨exp⟩  ) |
          ⟨cterm⟩

⟨cterm⟩ ⟼ ⟨term⟩ :X: ⟨cterm⟩ |
          POWERSET ( ⟨cterm⟩ ) |
          ⟨term⟩

⟨term⟩ ⟼ ⟨integer⟩ |
         ⟨character⟩|
         ⟨id⟩ |
         ⟨sqmemb⟩ |
         '|' ⟨exp⟩ '|' |
         ⟨funcall⟩ |
         ⟨set⟩ |
         ⟨seq⟩ |
         ⟨subset⟩ |
         ⟨sequence⟩

⟨sqmemb⟩ ⟼ ⟨id⟩ [ ⟨exp⟩ ]

⟨funcall⟩ ⟼ ⟨id⟩ ( ⟨actuals⟩ )

⟨actuals⟩ ⟼ ⟨exp⟩ , ⟨actuals⟩ |
             % ⟨actual_tail⟩

⟨actual_tail⟩ ⟼ , ⟨exp⟩ ⟨actual_tail⟩ |
                 ϵ

⟨i/o-stat⟩ ⟼ ⟨id⟩ = ⟨io-exp⟩ |
             ⟨seq⟩ = ⟨io-exp⟩

⟨io-exp ⟩ ⟼ INPUT ⟨string⟩ |
            OUTPUT ⟨string⟩

⟨sequence⟩ ⟼ SEQUENCE( ⟨vterm⟩ )

⟨subset⟩ ⟼ { ⟨arglist⟩ '|' ⟨body⟩ }

⟨set⟩ ⟼ { ⟨paramlist⟩ } |
        INTEGER |
        CHARACTER |
        EMPTY

⟨seq⟩ ⟼ << ⟨paramlist⟩ >>
```